IMS
Version 11

Application Programming

SC19-2428-05

IMS
Version 11

Application Programming

SC19-2428-05

Note
Before using this information and the product that it supports, be sure to read the general information under "Notices" on page 707.

This edition applies to IMS Version 11 (program number 5635-A02) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1974, 2015.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information ix
Prerequisite knowledge ix
IMS function names used in this information . . . ix
How new and changed information is identified . . x
How to read syntax diagrams x
Accessibility features for IMS Version 11. xii
How to send your comments xii

Changes to the IMS library xiii

Part 1. Application programming design 1

Chapter 1. Designing an application: Introductory concepts. 3
Storing and processing information in a database . . 3
Database hierarchy examples 5
Your program's view of the data 10
Processing a database record 12
Tasks for developing an application 13

Chapter 2. Designing an application: Data and local views. 15
An overview of application design 15
Identifying application data 17
Listing data elements 17
Naming data elements. 19
Documenting application data 20
Designing a local view 22
Analyzing data relationships 22
Local view examples 29

Chapter 3. Analyzing IMS application processing requirements 35
Defining IMS application requirements 35
Accessing databases with your IMS application program 36
Accessing data: the types of programs you can write for your IMS application 38
DB batch processing 39
TM batch processing 40
Processing messages: Message Processing Programs 40
Processing messages: IMS Fast Path Programs . . 41
Batch message processing: BMPs 42
Java message processing: JMPs 45
Java batch processing: JBPs 45
IMS programming integrity and recovery considerations 46
How IMS protects data integrity: commit points 46
Planning for program recovery: checkpoint and restart 49
Data availability considerations. 53
Use of STAE or ESTAE and SPIE in IMS programs 55
Dynamic allocation for IMS databases 56

Chapter 4. Analyzing CICS application processing requirements 57
Defining CICS application requirements 57
Accessing databases with your CICS application program 58
Writing a CICS program to access IMS databases . . 60
Writing a CICS online program 60
Using data sharing for your CICS program 61
Scheduling and terminating a PSB (CICS online programs only) 62
Linking and passing control to other programs (CICS online programs only) 62
How CICS distributed transactions access IMS . . 63
Maximizing the performance of your CICS system 63
Programming integrity and database recovery considerations for your CICS program 64
How IMS protects data integrity for CICS online programs 64
Recovering databases accessed by batch and BMP programs 64
Data availability considerations for your CICS program 69
Unavailability of a database 69
Unavailability of some data in a database . . . 70
The SETS or SETU and ROLS functions 70
Use of STAE or ESTAE and SPIE in IMS batch programs 71
Dynamic allocation for IMS databases 71

Chapter 5. Gathering requirements for database options 73
Analyzing data access 73
Direct access 74
Sequential access 78
Accessing z/OS files through IMS: GSAM . . . 80
Accessing IMS data through z/OS: SHSAM and SHISAM 80
Understanding how data structure conflicts are resolved 81
Using different fields: field-level sensitivity . . . 81
Resolving processing conflicts in a hierarchy: secondary indexing. 82
Creating a new hierarchy: logical relationships . 86
Providing data security 91
Keeping a program from accessing the data: data sensitivity 91
Preventing a program from updating data: processing options 93
Read without integrity. 95

© Copyright IBM Corp. 1974, 2015

Chapter 6. Gathering requirements for message processing options 99
Identifying online security requirements 99
Analyzing screen and message formats 101
An overview of MFS 102
An overview of basic edit 102
Editing considerations in your application . . . 102
Gathering requirements for conversational processing 103
What happens in a conversation 104
Designing a conversation 104
Important points about the scratchpad area (SPA) 105
Recovery considerations in conversations . . . 105
Identifying output message destinations 106
The originating terminal 107
To other programs and terminals 107

Chapter 7. Designing an application for APPC 111
Overview of APPC and LU 6.2 111
Application program types 111
Application objectives 113
Conversation type 114
Conversation state 115
Synchronization level 115
Introduction to resource recovery 116
Summary of z/OS Resource Recovery Services support 119
Distributed sync point 120
Application programming interface for LU type 6.2 121
LU 6.2 partner program design 122
LU 6.2 flow diagrams 122
Integrity tables 142
DFSAPPC message switch 144

Chapter 8. Testing an IMS application program 147
Recommendations for testing an IMS program . . 147
Testing DL/I call sequences (DFSDDLT0) before testing your IMS program 147
Using BTS to test your IMS program 148
Tracing DL/I calls with image capture for your IMS program 148
Using image capture with DFSDDLT0 149
Restrictions on using image capture output . . 149
Running image capture online 149
Running image capture as a batch job 150
Retrieving image capture data from the log data set 150
Requests for monitoring and debugging your IMS program 151
Retrieving database statistics: the STAT call . . 151
Writing Information to the system log: the LOG request 165
What to do when your IMS program terminates abnormally 165

Chapter 9. Testing a CICS application program 169
Recommendations for testing a CICS program . . 169
Testing your CICS program 169
Tracing DL/I calls with image capture 170
Requests for monitoring and debugging your CICS program 174
What to do when your CICS program terminates abnormally 174

Chapter 10. Documenting your application program 177
Documentation for other programmers 177
Documentation for end users 177

Part 2. Application programming for IMS DB 179

Chapter 11. Writing your application programs for IMS DB 181
Programming guidelines 181
Segment search arguments (SSAs) 182
SSA guidelines 185
Multiple qualification statements 186
SSAs and command codes 189
Considerations for coding DL/I calls and data areas 190
Preparing to run your CICS DL/I call program . . 192
Examples of how to code DL/I calls and data areas 192
Coding a batch program in assembler language 192
Coding a CICS online program in assembler language 194
Coding a batch program in C language 196
Coding a batch program in COBOL 198
Coding a CICS online program in COBOL . . . 201
Coding a program in Java 205
Coding a batch program in Pascal 205
Coding a batch program in PL/I 207
Coding a CICS online program in PL/I 210

Chapter 12. Defining application program elements for IMS DB 213
Formatting DL/I calls for language interfaces . . 213
Assembler language application programming . . 213
C language application programming 215
COBOL application programming 218
Java application programming for IMS 221
Pascal application programming 221
Application programming for PL/I 224
Specifying the I/O PCB mask 226
Specifying the DB PCB mask 230
Specifying the AIB mask 232
Specifying the AIB mask for ODBA applications 234
Specifying the UIB (CICS online programs only) 237
Specifying the I/O areas 239
Formatting segment search arguments (SSAs) . . 240
SSA coding rules 240
SSA coding formats 241

Data areas in GSAM databases 245
AIBTDLI interface 245
Language specific entry points 246
Program communication block (PCB) lists 249
The AERTDLI interface 251
Language environments 251
Special DL/I situations for IMS DB programming 252

Chapter 13. Establishing a DL/I interface from COBOL or PL/I 255

Chapter 14. Current position in the database after each call 257
Current position after successful calls 257
Position after retrieval calls 259
Position after DLET 259
Position after REPL 261
Position after ISRT. 261
Current position after unsuccessful calls 263
Multiple processing 267
Advantages of using multiple positioning . . . 270
Multiple DB PCBs 273

Chapter 15. Using IMS application program sync points 275
Commit process 275
Two-phase commit in the synchronization process 276
Unit of recovery 278
DBCTL single-phase commit 279
Sync-point log records 279
Sync points with a data-propagation manager . . 280

Chapter 16. Recovering databases and maintaining database integrity . . 283
Issuing checkpoints 283
Restarting your program from the latest checkpoint 284
Maintaining database integrity (IMS batch, BMP, and IMS online regions) 284
Backing out to a prior commit point: ROLL, ROLB, and ROLS 284
Backing out to an intermediate backout point: SETS, SETU, and ROLS 288
Reserving segments for the exclusive use of your program 291

Chapter 17. Secondary indexing and logical relationships 293
How secondary indexing affects your program . . 293
SSAs with secondary indexes 293
Multiple qualification statements with secondary indexes 294
DL/I returns with secondary indexes 296
Status codes for secondary indexes 296
Processing segments in logical relationships . . . 297
How logical relationships affect your programming 299
Status codes for logical relationships 299

Chapter 18. HALDB selective partition processing 301

Chapter 19. Processing GSAM databases 305
Accessing GSAM databases. 305
PCB masks for GSAM databases 305
Retrieving and inserting GSAM records . . . 308
Explicit open and close calls to GSAM 309
GSAM record formats 310
GSAM I/O areas 310
GSAM status codes 311
Symbolic CHKP and XRST with GSAM 312
GSAM coding considerations 312
Origin of GSAM data set characteristics 313
DD statement DISP parameter for GSAM data sets 314
Extended checkpoint restart for GSAM data sets 315
Concatenated data sets used by GSAM 316
Specifying GSAM data set attributes. 316
DLI, DBB, and BMP region types and GSAM 317

Chapter 20. Processing Fast Path databases 319
Fast Path database calls 320
Main storage databases (MSDBs) 321
Restrictions on using calls for MSDBs 321
Data entry databases (DEDBs) 322
Updating segments: REPL, DLET, ISRT, and FLD 322
Checking the contents of a field: FLD/VERIFY 323
Changing the contents of a field: FLD/CHANGE. 325
Example of using FLD/VERIFY and FLD/CHANGE. 326
Commit-point processing in MSDBs and DEDBs 327
Processing DEDBs (IMS and CICS with DBCTL) 328
Processing Fast Path DEDBs with subset pointer command codes 328
Retrieving location with the POS call (for DEDB only) 333
Commit-point processing in a DEDB 336
P processing option 336
H processing option 336
Calls with dependent segments for DEDBs . . . 337
DEDB DL/I calls to extract DEDB information . . 338
AL_LEN Call 342
DI_LEN Call 343
DS_LEN Call 343
AREALIST Call. 343
DEDBINFO Call 344
DEDSTR Call 344
Fast Path coding considerations 345

Chapter 21. Writing ODBA application programs 347
General application program flow of ODBA application programs 347
Server program structure 350
DB2 for z/OS stored procedures use of ODBA 351

Testing an ODBA application program 352
Tracing DL/I calls with image capture to test your ODBA program 353
Using image capture with DFSDDLT0 to test your ODBA program 354
Running image capture online. 354
Retrieving image capture data from the log data set 355
Requests for monitoring and debugging your ODBA program. 355
What to do when your ODBA program terminates abnormally 356

Chapter 22. Programming with the IMS support for DRDA 359
DDM commands for data operations with the IMS support for DRDA 360

Part 3. Application programming for IMS TM 363

Chapter 23. Defining application program elements for IMS TM 365
Formatting DL/I calls for language interfaces . . 365
Application programming for assembler language 365
Application programming for C language 368
Application programming for COBOL 370
Java application programming for IMS 373
Application programming for Pascal 373
Application programming for PL/I 376
Relationship of calls to PCB types 378
Specifying the I/O PCB mask 379
Specifying the alternate PCB mask 383
Specifying the AIB mask 383
Specifying the I/O areas. 385
AIBTDLI interface 386
Specifying language-specific entry points 387
Program communication block (PCB) lists 389
Language environments 390
Special DL/I situations for IMS TM programming 391

Chapter 24. Message processing with IMS TM 393
How your program processes messages 393
Message types 393
When a message is processed 396
Results of a message: I/O PCB 398
How IMS TM edits messages 398
Printing output messages 399
Using Basic Edit 399
Using Intersystem Communication Edit . . . 400
Using Message Format Service 400
Using LU 6.2 User Edit exit routine (optional) 407
Message processing considerations for DB2 . . . 407
Sending messages to other terminals and programs 408
Sending messages to other terminals 409
Sending messages to other IMS application programs 411
How the VTAM I/O facility affects your VTAM terminal 413
Communicating with other IMS TM systems using Multiple Systems Coupling. 413
Implications of MSC for program coding . . . 413
Receiving messages from other IMS TM systems 414
Sending messages to alternate destinations in other IMS TM systems 415
IMS conversational processing. 416
A conversational example 417
Conversational structure. 418
Replying to the terminal. 422
Conversational processing using ROLB, ROLL, and ROLS 422
Passing the conversation to another conversational program 423
Message switching in APPC conversations . . 425
Processing conversations with APPC 427
Ending the APPC conversation 427
Coding a conversational program 428
Standard IMS application programs 428
Modified IMS application programs 429
CPI-C driven application programs 429
Processing conversations with OTMA 430
Backing out to a prior commit point: ROLL, ROLB, and ROLS calls 431
Comparison of ROLB, ROLL, and ROLS . . . 431
ROLL 432
ROLB 432
ROLS 434
Backing out to an intermediate backout point: SETS/SETU and ROLS 434
Writing message-driven programs 437
Coding DC calls and data areas 438
Before coding your program 438
MPP code examples 439
Message processing considerations for DB2 . . 445

Chapter 25. IMS Spool API. 447
Managing the IMS Spool API overall design . . . 447
IMS Spool API design 447
Sending data to the JES spool data sets. . . . 448
IMS Spool API performance considerations . . 448
IMS Spool API application coding considerations 449
Understanding parsing errors 452
Diagnosis examples 453
Understanding allocation errors 456
Understanding dynamic output for print data sets 456
Sample programs using the Spool API 457

Chapter 26. IMS Message Format Service 461
Advantages of using MFS 461
MFS control blocks 462
MFS examples 463
Relationship between MFS control blocks and screen format 467
Overview of MFS components. 468
Devices and logical units that operate with MFS 470

Using distributed presentation management (DPM) 472

Chapter 27. Callout requests for services or data 473
Callout request approaches 474
Resume tpipe protocol 475
Implementing the synchronous callout function 475
Implementing the asynchronous callout function 478

Part 4. Application programming for EXEC DLI 481

Chapter 28. Writing your application programs for EXEC DLI 483
Programming guidelines 483
Coding a program in assembler language . . . 484
Coding a program in COBOL 488
Coding a program in PL/I 491
Coding a program in C 495
Preparing your EXEC DLI program for execution 501
Translator, compiler, and binder options required for EXEC DLI 501

Chapter 29. Defining application program elements 503
Specifying an application interface block (AIB) . . 503
Specifying the DL/I interface block (DIB) 503
Defining a key feedback area 507
Defining I/O areas 507

Chapter 30. EXEC DLI commands for an application program 509
PCBs and PSB 509

Chapter 31. Recovering databases and maintaining database integrity . . 513
Issuing checkpoints in a batch or BMP program 513
Restarting your program and checking for position 514
Backing out database updates dynamically: the ROLL and ROLB commands 514
Using intermediate backout points: the SETS and ROLS commands 514

Chapter 32. Processing Fast Path databases 517
Processing Fast Path DEDBs with subset pointer options 517
Preparing to use subset pointers 519
Designating subset pointers 520
Subset pointer options 520
Subset pointer status codes 527
The POS command 528
Locating a specific sequential dependent segment 528
Locating the last inserted sequential dependent segment 529
Identifying free space with the POS command 529
The P processing option 530

Chapter 33. Comparing command-level and call-level programs 531
DL/I calls for IMS and CICS 531
Comparing EXEC DLI commands and DL/I calls 531
Comparing command codes and options 533

Chapter 34. Data availability enhancements 535

Part 5. Java application development for IMS 537

Chapter 35. IMS solutions for Java development overview 539

Chapter 36. Comparison of hierarchical and relational databases . 543

Chapter 37. Programming with the IMS Universal drivers 549
IMS Universal drivers overview 549
Distributed and local connectivity with the IMS Universal drivers 550
Comparison of IMS Universal drivers programming approaches for accessing IMS . . 553
Support for variable-length database segments with the IMS Universal drivers 555
Generating the runtime Java metadata class . . 557
Hospital database example 557
Programming using the IMS Universal Database resource adapter 560
Overview of the IMS Universal Database resource adapter 561
Transaction types and programming interfaces supported by the IMS Universal Database resource adapter 561
Connecting to IMS with the IMS Universal Database resource adapter 563
Sample EJB application using the IMS Universal Database resource adapter CCI programming interface 570
Accessing IMS data with the DLIInteractionSpec class 572
Accessing IMS data with the SQLInteractionSpec class 576
Accessing IMS data with the IMS Universal JCA/JDBC driver 579
Programming with the IMS Universal JDBC driver 581
Supported drivers for JDBC 582
Connecting to IMS using the IMS Universal JDBC driver 583
Sample application for the IMS Universal JDBC driver 591
Writing SQL queries to access an IMS database with the IMS Universal JDBC driver. 592

IMS Universal JDBC driver support for XML 605
Data transformation support for JDBC 610
Programming with the IMS Universal DL/I driver 615
Basic steps in writing a IMS Universal DL/I driver application 615
Java packages for IMS Universal DL/I driver support 616
Connecting to an IMS database by using the IMS Universal DL/I driver 616
IMS Universal DL/I driver interfaces for executing DL/I operations 618
Inspecting the PCB status code and related information using the com.ibm.ims.dli.AIB interface 634
Committing or rolling back DL/I transactions 635
Configuring the IMS Universal drivers for SSL support 637
Configuring the IMS Universal Database resource adapter for SSL support in a container-managed environment 638
Configuring IMS Universal drivers for SSL support in a stand-alone environment 638
Tracing IMS Universal drivers applications . . . 639

Chapter 38. Programming Java dependent regions 641
Overview of the IMS Java dependent regions. . . 641
Programming with the IMS Java dependent region resource adapter 642
Developing JMP applications with the IMS Java dependent region resource adapter 643
Developing JBP applications with the IMS Java dependent region resource adapter 652
Issuing synchronous callout requests from a Java dependent region 660
Program switching in JMP and JBP applications 662
IBM Enterprise COBOL for z/OS interoperability with JMP and JBP applications 665
IBM Enterprise COBOL for z/OS backend applications in a JMP or JBP region 666
IBM Enterprise COBOL for z/OS frontend applications in a JMP or JBP region 667
Accessing DB2 for z/OS databases from JMP or JBP applications 667

Chapter 39. Programming with the classic Java APIs for IMS 669
Programming enterprise Java applications with classic Java APIs for IMS resource adapters . . . 669
Accessing IMS data from WebSphere Application Server for z/OS with the classic Java APIs for IMS 669
Bean-managed EJB programming model . . . 670
Container-managed EJB programming model 672
Servlet programming model 672
Requirements for WebSphere Application Server for z/OS with the classic Java APIs for IMS . . 673
Deployment descriptor requirements for the classic Java APIs for IMS 674
Programming Java applications in DB2 for z/OS stored procedures with the classic Java APIs for IMS 674
Accessing IMS data from DB2 for z/OS stored procedures using the classic Java APIs for IMS . 675
DB2 for z/OS stored procedures programming model with the classic Java APIs for IMS . . . 675
Programming Java applications for CICS with the classic Java APIs for IMS 676
Accessing IMS data from CICS with the classic Java APIs for IMS 676
CICS programming model with the classic Java APIs for IMS 677
Programming with the IMS classic JDBC driver 678
Sample PSB, DBD and DLIModel IMS Java report 678
Data transformation support for JDBC 680
Connections to IMS databases 686
JDBC interfaces supported by the IMS classic JDBC driver 686
SQL keywords and extensions for the IMS classic JDBC driver 688
Sample application that uses the IMS classic JDBC driver 698
Problem determination for Java applications . . . 699
Exceptions thrown from IMS DL/I calls . . . 699
XML tracing for the classic Java APIs for IMS 700

Part 6. Appendixes 705

Notices 707
Programming interface information 709
Privacy policy considerations 710

Bibliography 711

Index 713

About this information

These topics provide guidance information for writing application programs that access IMS™ databases or IMS transactions. The topics describe how to gather and analyze program requirements, and how to develop and debug IMS application programs. They also describe how to use different programming languages to issue DL/I calls, and include information about the IMS solutions for SQL and Java™ development. They also describe how to use different programming languages to issue EXEC DL/I calls. Application programming interface (API) information is in *IMS Version 11 Application Programming APIs*.

This information is available in IBM® Knowledge Center at www.ibm.com/support/knowledgecenter.

Prerequisite knowledge

This information is a guide to IMS application programming for any of the following environments:

- IMS Database Manager (IMS DB), including IMS Database Control (DBCTL)
- IMS Transaction Manager (IMS TM)
- CICS® EXEC DLI
- WebSphere® Application Server for z/OS®
- WebSphere Application Server for distributed platforms
- Java dependent regions (JMP and JBP)
- Any environment for stand-alone Java application development

This book provides guidance information for writing application programs that access IMS databases or process IMS messages. It also describes how to use different programming languages to make DL/I, EXEC DLI, or JDBC calls that interact with IMS. API (application programming interface) information is in *IMS Version 11 Application Programming APIs*.

You can learn more about z/OS by visiting the "z/OS basic skills education" topics in IBM Knowledge Center.

You can gain an understanding of basic IMS concepts by reading *An Introduction to IMS*, an IBM Press publication. An excerpt from this publication is available in the IBM Knowledge Center.

IBM offers a wide variety of classroom and self-study courses to help you learn IMS. For a complete list of courses available, go to the IMS home page at ibm.com/ims and link to the Training and Certification page.

IMS function names used in this information

In this information, the term HALDB Online Reorganization refers to the integrated HALDB Online Reorganization function that is part of IMS Version 11, unless otherwise indicated.

IMS provides an integrated IMS Connect function, which offers a functional replacement for the IMS Connect tool (program number 5655-K52). In this

© Copyright IBM Corp. 1974, 2015

information, the term IMS Connect refers to the integrated IMS Connect function that is part of IMS Version 11, unless otherwise indicated.

How new and changed information is identified

New and changed information in most IMS library PDF publications is denoted by a character (revision marker) in the left margin. The first edition (-00) of *Release Planning*, as well as the *Program Directory* and *Licensed Program Specifications*, do not include revision markers.

Revision markers follow these general conventions:

- Only technical changes are marked; style and grammatical changes are not marked.
- If part of an element, such as a paragraph, syntax diagram, list item, task step, or figure is changed, the entire element is marked with revision markers, even though only part of the element might have changed.
- If a topic is changed by more than 50%, the entire topic is marked with revision markers (so it might seem to be a new topic, even though it is not).

Revision markers do not necessarily indicate all the changes made to the information because deleted text and graphics cannot be marked with revision markers.

How to read syntax diagrams

The following rules apply to the syntax diagrams that are used in this information:

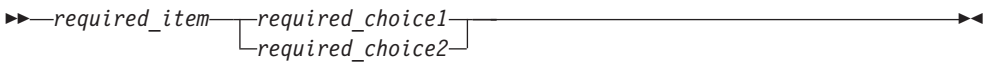
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following conventions are used:
  - The >>--- symbol indicates the beginning of a syntax diagram.
  - The ---> symbol indicates that the syntax diagram is continued on the next line.
  - The >--- symbol indicates that a syntax diagram is continued from the previous line.
  - The --->< symbol indicates the end of a syntax diagram.
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

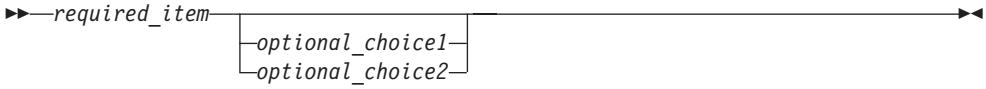
  If an optional item appears above the main path, that item has no effect on the execution of the syntax element and is used only for readability.

- If you can choose from two or more items, they appear vertically, in a stack.

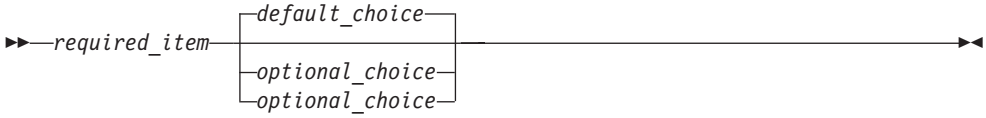
  If you *must* choose one of the items, one item of the stack appears on the main path.



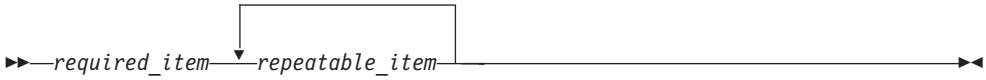
If choosing one of the items is optional, the entire stack appears below the main path.



If one of the items is the default, it appears above the main path, and the remaining choices are shown below.



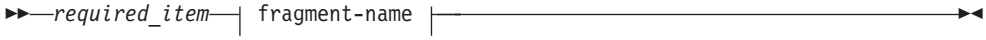
- An arrow returning to the left, above the main line, indicates an item that can be repeated.



If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.



fragment-name:

- In IMS, a b symbol indicates one blank position.
- Keywords, and their minimum abbreviations if applicable, appear in uppercase. They must be spelled exactly as shown. Variables appear in all lowercase italic letters (for example, *column-name*). They represent user-supplied names or values.
- Separate keywords and parameters by at least one space if no intervening punctuation is shown in the diagram.

- Enter punctuation marks, parentheses, arithmetic operators, and other symbols, exactly as shown in the diagram.
- Footnotes are shown by a number in parentheses, for example (1).

Accessibility features for IMS Version 11

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in z/OS products, including IMS Version 11. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size.

Keyboard navigation

You can access IMS Version 11 ISPF panel functions by using a keyboard or keyboard shortcut keys.

For information about navigating the IMS Version 11 ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide Volume 1*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

Online documentation for IMS Version 11 is available in IBM Knowledge Center.

IBM and accessibility

See the *IBM Human Ability and Accessibility Center* at www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments

Your feedback is important in helping us provide the most accurate and highest quality information. If you have any comments about this or any other IMS information, you can take one of the following actions:

- Click the **Feedback** link at the bottom of any IBM Knowledge Center topic.
- Send an email to imspubs@us.ibm.com. Be sure to include the book title and the publication number.

Changes to the IMS library

Changes to the IMS library for IMS Version 11 include minor title changes to all the publications, merging four sets of publications, a change to how *IMS Messages and Codes* are delivered, and a reorganization of *IMS Version 11 Exit Routines*. The library is no longer provided in BookManager format. In addition, as of the May, 2012, a master index is no longer provided.

The titles of all PDF publications in the IMS Version 11 library have been changed so that the publication titles match the contents view (navigation tree) titles in IBM Knowledge Center. For example, the *IMS Diagnosis Guide* has been renamed to *IMS Diagnosis*. In addition, the titles for the three *IMS Command Reference* PDF volumes have been renamed so that their titles better reflect the contents of each publication, as follows:

- *IMS Version 11 Commands, Volume 1: IMS Commands A-M*
- *IMS Version 11 Commands, Volume 2: IMS Commands N-V*
- *IMS Version 11 Commands, Volume 3: IMS Component and z/OS Commands*

Four sets of publications have been streamlined:

- To eliminate redundancy and group relevant application programming planning information with associated application programming tasks, the *IMS Application Programming Planning Guide* has been merged with the *IMS Application Programming Guide*, in a new title: *IMS Version 11 Application Programming*.
- To group diagnosis reference information with associated diagnosis tasks, the *IMS Diagnosis Reference* has been merged with the *IMS Diagnosis Guide*, in a new title: *IMS Version 11 Diagnosis*.
- To group all system administration tasks, regardless of whether the tasks relate to using a single IMS or an IMSplex, the *IMS IMSplex Administration Guide* has been merged with the *IMS System Administration Guide*, in a new title: *IMS Version 11 System Administration*.
- To group system definition reference information with associated system definition tasks, the *IMS System Definition Reference* has been merged with the *IMS System Definition Guide*, in a new title: *IMS Version 11 System Definition*.

IMS Version 11 Exit Routines is now organized by the call interface and available services for each group of exit routines, as follows:

- IMS system exits, database manager exits, and transaction manager exit routines are grouped under “IMS control region exit routines.”These exit routines use IMS interfaces and callable services, as described in the topic “Overview of exit routines for IMS.”
- Exit routines that run in an address space with BPE support, including those for CQS, BPE-based DBRC, and most CSL components, are grouped under “Base Primitive Environment-based exit routines.” BPE-based exit routines use the interfaces and services provided by the Base Primitive Environment, as described in the topic “BPE user-supplied exit routine interfaces and services.”
- The CQS exit routines are now divided into two groups: those that run in the CQS address space and have access to BPE services are grouped under “BPE-based CQS user-supplied exit routines,” and those that execute in a CQS client address space are grouped under “CQS client exit routines.”

© Copyright IBM Corp. 1974, 2015

- Standard DBRC exit routines are grouped under “IMS system exit routines,” while the variants that run in BPE are group under “BPE-based DBRC exit routines.”
- CSL SCI exit routines are now divided into two categories: those that run in the CSL SCI address space are grouped under “BPE-based SCL SCI user exit routines,” and those that run in the address space of a registered IMSplex member are grouped under “CSL SCI IMSPlex member exit routines.”
- Exit routines that have specific guidelines, including those for IMS Connect, CQS clients, and CSL SCI IMSPlex members, are described in the sections for those routines.

Part 1. Application programming design

To design an application program for IMS, you need to identify the application data and analyze requirements for application processing. You may also need to perform other tasks, such as gathering requirements for database and message processing options, and testing an application program.

© Copyright IBM Corp. 1974, 2015

Chapter 1. Designing an application: Introductory concepts

This section provides an introduction to designing application programs. It explains some basic concepts about processing a database, and gives an overview of the tasks covered in this information.

Storing and processing information in a database

The advantages of storing and processing data in a database are that all of the data needs to appear only once and that each program must process only the data that it needs.

One way to understand this is to compare three ways of storing data: in separate files, in a combined file, and in a database.

Storing data in separate files

If you keep separate files of data for each part of your organization, you can ensure that each program uses only the data it needs, but you must store a lot of data in multiple places simultaneously. Problems with keeping separate files are:

- Redundant data takes up space that could be put to better use
- Maintaining separate files can be difficult and complex

For example, suppose that a medical clinic keeps separate files for each of its departments, such as the clinic department, the accounting department, and the ophthalmology department:

- The clinic department keeps data about each patient who visits the clinic, such as:
  - Identification number
  - Name
  - Address
  - Illnesses
  - Date of each illness
  - Date patient came to clinic for treatment
  - Treatment given for each illness
  - Doctor that prescribed treatment
  - Charge for treatment
- The accounting department also keeps information about each patient. The information that the accounting department might keep for each patient is:
  - Identification number
  - Name
  - Address
  - Charge for treatment
  - Amount of payments
- The information that the ophthalmology department might keep for each patient is:
  - Identification number

© Copyright IBM Corp. 1974, 2015

Name
Address
Illnesses relating to ophthalmology
Date of each illness
Names of members in patient's household
Relationship between patient and each household member

If each of these departments keeps separate files, each department uses only the data that it needs, but much of the data is redundant. For example, every department in the clinic uses at least the patient's number, name, and address. Updating the data is also a problem, because if a department changes a piece of data, the same data must be updated in each separate file. Therefore, it is difficult to keep the data in each department's files current. Current data might exist in one file while defunct data remains in another file.

Storing data in a combined file

Another way to store data is to combine all the files into one file for all departments to use. In the medical example, the patient record that would be used by each department would contain these fields:

Identification number
Name
Address
Illnesses
Date of each illness
Date patient came to clinic for treatment
Treatment given for each illness
Doctor that prescribed treatment
Charge for treatment
Amount of payments
Names of members in patient's household
Relationship between patient and each household member

Using a combined file solves the updating problem, because all the data is in one place, but it creates a new problem: the programs that process this data must access the entire file record to get to the part that they need. For example, to process only the patient's number, charges, and payments, an accounting program must access all of the other fields also. In addition, changing the format of any of the fields within the patient's record affects all the application programs, not just the programs that use that field.

Using combined files can also involve security risks, because all of the programs have access to all of the fields in a record.

Storing data in a database

Storing data in a database gives you the advantages of both separate files and combined files: all the data appears only once, and each program has access to the data that it needs. This means that:

- When you update a field, you do it in one place only.

- Because you store each piece of information only in one place, you cannot have an updated version of the information in one place and an out-of-date version in another place.
- Each program accesses only the data it needs.
- You can prevent programs from accessing private or secured information.

In addition, storing data in a database has two advantages that neither of the other ways has:

- If you change the format of part of a database record, the change does not affect the programs that do not use the changed information.
- Programs are not affected by how the data is stored.

Because the program is independent of the physical data, a database can store all the data only once and yet make it possible for each program to use only the data that it needs. In a database, what the data looks like when it is stored is different from what it looks like to an application program.

Database hierarchy examples

In an IMS DB, a record is stored and accessed in a hierarchy. A hierarchy shows how each piece of data in a record relates to other pieces of data in the record.

IMS connects the pieces of information in a database record by defining the relationships between the pieces of information that relate to the same subject. The result is a database hierarchy.

Medical hierarchy example

The medical database shown in following figure contains information that a medical clinic keeps about its patients. The hierarchies used in the medical hierarchy example are used with full-function databases and Fast Path data entry databases (DEDBs).

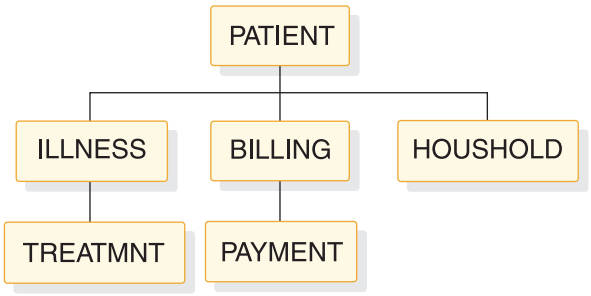


Figure 1. Medical hierarchy

Each piece of data represented in the figure above is called a *segment* in the hierarchy. Each segment contains one or more *fields* of information. The PATIENT segment, for example, contains all the information that relates strictly to the patient: the patient's identification number, name, and address.

Definitions: A *segment* is the smallest unit of data that an application program can retrieve from the database. A *field* is the smallest unit of a segment.

The PATIENT segment in the medical database is the *root segment*. The segments below the root segment are the *dependents*, or children, of the root. For example,

ILLNESS, BILLING, and HOUSHOLD are all children of PATIENT. ILLNESS, BILLING, and HOUSHOLD are called direct dependents of PATIENT; TREATMNT and PAYMENT are also dependents of PATIENT, but they are not direct dependents, because they are at a lower level in the hierarchy.

A *database record* is a single root segment (root segment *occurrence*) and all of its dependents. In the medical example, a database record is all of the information about one patient.

Definitions: A *root segment* is the highest-level segment. A dependent is a segment below a root segment. A root segment occurrence is a database record and all of its dependents.

Each database record has only one root segment occurrence, but it might have several occurrences at lower levels. For example, the database record for a patient contains only one occurrence of the PATIENT segment type, but it might contain several ILLNESS and TREATMNT segment occurrences for that patient.

The tables that follow show the layouts of each segment in the hierarchy.

The segment's field names are in the first row of each table. The number below each field name is the length in bytes that has been defined for that field.

- **PATIENT Segment**

  The following table shows the PATIENT segment.

  It has three fields:

  – The patient's number (PATNO)
  – The patient's name (NAME)
  – The patient's address (ADDR)

  PATIENT has a unique key field: PATNO. PATIENT segments are stored in ascending order based on the patient number. The lowest patient number in the database is 00001 and the highest is 10500.

Table 1. PATIENT segment

Field name	Field length
PATNO	10
NAME	5
ADDR	30

- **ILLNESS Segment**

  The following figure shows the ILLNESS segment.

  It has two fields:

  – The date when the patient came to the clinic with the illness (ILLDATE)
  – The name of the illness (ILLNAME)

  The key field is ILLDATE. Because it is possible for a patient to come to the clinic with more than one illness on the same date, this key field is non-unique, that is, there may be more than one ILLNESS segment with the same (an equal) key field value.

  Usually during installation, the database administrator (DBA) decides the order in which to place the database segments with equal or no keys. The DBA can use the RULES keyword of the SEGM statement of the DBD to specify the order of the segments.

For segments with equal keys or no keys, RULES determines where the segment is inserted. Where RULES=LAST, ILLNESS segments that have equal keys are stored on a first-in-first-out basis among those with equal keys. ILLNESS segments with unique keys are stored in ascending order on the date field, regardless of RULES. ILLDATE is specified in the format YYYYMMDD.

Table 2. ILLNESS segment

Field name	Field length
ILLDATE	8
ILLNAME	10

- **TREATMNT Segment**

  The following table shows the TREATMNT segment.

  It contains four fields:

  – The date of the treatment (DATE)
  – The medicine that was given to the patient (MEDICINE)
  – The quantity of the medicine that the patient received (QUANTITY)
  – The name of the doctor who prescribed the treatment (DOCTOR)

  The TREATMNT segment's key field is DATE. Because a patient may receive more than one treatment on the same date, DATE is a non-unique key field. TREATMNT, like ILLNESS, has been specified as having RULES=LAST. TREATMNT segments are also stored on a first-in-first-out basis. DATE is specified in the same format as ILLDATE—YYYYMMDD.

Table 3. TREATMNT segment

Field name	Field length
DATE	8
MEDICINE	10
QUANTITY	4
DOCTOR	10

- **BILLING Segment**

  The following table shows the BILLING segment. It has only one field: the amount of the current bill. BILLING has no key field.

Table 4. BILLING segment

Field name	Field length
BILLING	6

- **PAYMENT Segment**

  The following table shows the PAYMENT segment. It has only one field: the amount of payments for the month. The PAYMENT segment has no key field.

Table 5. PAYMENT segment

Field name	Field length
PAYMENT	6

- **HOUSHOLD Segment**

  The following table shows the HOUSHOLD segment.

  It contains two fields:

– The names of the members of the patient's household (RELNAME)
– How each member of the household is related to the patient (RELATN)

The HOUSHOLD segment's key field is RELNAME.

Table 6. HOUSHOLD segment

Field name	Field length
RELNAME	10
RELATN	8

Bank account hierarchy example

The bank account hierarchy is an example of an application program that is used with main storage databases (MSDBs). In the medical hierarchy example, the database record for a particular patient comprises the PATIENT segment and all of the segments underneath the PATIENT segment. In an MSDB, such as the one in the bank account example, the segment is the whole database record. The database record contains only the fields that the segment contains.

The two types of MSDBs are *related* and *nonrelated*. In related MSDBs, each segment is "owned" by one logical terminal. The "owned" segment can only be updated by the terminal that owns it. In nonrelated MSDBs, the segments are not owned by logical terminals. The following examples of a related MSDB and a nonrelated MSDB illustrate the differences between the two types of databases.

Related MSDBs

Related MSDBs can be fixed or dynamic. In a fixed related MSDB, you can store summary data about a particular teller at a bank. For example, you can have an identification code for the teller's terminal. Then you can keep a count of that teller's transactions and balance for the day. This type of application requires a segment with three fields:

TELLERID
A two-character code that identifies the teller

TRANCNT
The number of transactions the teller has processed

TELLBAL
The balance for the teller

The following table shows what the segment for this type of application program looks like.

Table 7. Teller segment in a fixed related MSDB

TELLERID	TRANCNT	TELLBAL

Some of the characteristics of fixed related MSDBs include:

- You can only read and replace segments. You cannot delete or insert segments. In the bank teller example, the teller can change the number of transactions processed, but you cannot add or delete any segments. You never need to add or delete segments.
- Each segment is assigned to one logical terminal. Only the owning terminal can change a segment, but other terminals can read the segment. In the bank teller

example, you do not want tellers to update the information about other tellers, but you allow the tellers to view each other's information. Tellers are responsible for their own transactions.

- The name of the logical terminal that owns the segment is the segment's key. Unlike non-MSDB segments, the MSDB key is not a field of the segment. It is used as a means of storing and accessing segments.
- A logical terminal can only own one segment in any one MSDB.

In a dynamic related MSDB, you can store data summarizing the activity of all bank tellers at a single branch. For example, this segment contains:

BRANCHNO
The identification number for the branch

TOTAL
The bank branch's current balance

TRANCNT
The number of transactions for the branch on that day

DEPBAL
The deposit balance, giving the total dollar amount of deposits for the branch

WTHBAL
The withdrawal balance, giving the dollar amount of the withdrawals for the branch

The following table shows what the branch summary segment looks like in a dynamic related MSDB.

Table 8. Branch summary segment in a dynamic related MSDB

BRANCHNO	TOTAL	TRANCNT	DEPBAL	WTHBAL

How dynamic related MSDBs differ from fixed related MSDBs:

- The owning logical terminal can delete and insert segments in a dynamic related MSDB.
- The MSDB can have a pool of unassigned segments. This kind of segment is assigned to a logical terminal when the logical terminal inserts it, and is returned to the pool when the logical terminal deletes it.

Nonrelated MSDBs

A nonrelated MSDB is used to store data that is updated by several terminals during the same time period. For example, you might store data about an individuals' bank accounts in a nonrelated MSDB segment, so that the information can be updated by a teller at any terminal. Your program might need to access the data in the following segment fields:

ACCNTNO
The account number

BRANCH
The name of the branch where the account is

TRANCNT
The number of transactions for this account this month

BALANCE
The current balance

The following table shows what the account segment in a nonrelated MSDB application program looks like.

Table 9. Account segment in a nonrelated MSDB

ACCNTNO	BRANCH	TRANCNT	BALANCE

The characteristics of nonrelated MSDBs include:

- Segments are not owned by terminals as they are in related MSDBs. Therefore, IMS programs and Fast Path programs can update these segments. Updating segments is not restricted to the owning logical terminal.
- Your program cannot delete or insert segments.
- Segment keys can be the name of a logical terminal. A nonrelated MSDB exists with terminal-related keys. The segments are not owned by the logical terminals, and the logical terminal name is used to identify the segment.
- If the key is not the name of a logical terminal, it can be any value, and it is in the first field of the segment. Segments are loaded in key sequence.

Your program's view of the data

IMS uses two kinds of control blocks to enable application programs to be independent of your method of storing data in the database, the database description (DBD), and the database program communication block (DB PCB).

Database Description (DBD)

A *database description (DBD)* is physical structure of the database. The DBD also defines the appearance and contents, or fields, that make up each of the segment types in the database.

For example, the DBD for the medical database hierarchy shown in "Medical hierarchy example" describes the physical structure of the hierarchy and each of the six segment types in the hierarchy: PATIENT, ILLNESS, TREATMNT, BILLING, PAYMENT, and HOUSHOLD.

Related Reading: For more information on generating DBDs, see *IMS Version 11 Database Utilities*.

Database Program Communication Block (DB PCB)

A *database program communication block (DB PCB)* is a control block that defines an application program's view of the database. An application program often needs to process only some of the segments in a database. A PCB defines which of the segments in the database the program is allowed to access—which segments the program is sensitive to.

The data structures that are available to the program contain only segments that the program is sensitive to. The PCB also defines how the application program is allowed to process the segments in the data structure: whether the program can only read the segments, or whether it can also update them.

To obtain the highest level of data availability, your PCBs should request the fewest number of sensitive segments and the least capability needed to complete the task.

All the DB PCBs for a single application program are contained in a *program specification block* (PSB). A program might use only one DB PCB (if it processes only one data structure) or it might use several DB PCBs, one for each data structure.

Related Reading: For more information on generating PSBs, see *IMS Version 11 Database Utilities*.

The following figure illustrates the concept of defining a view for an application program. An accounting program that calculates and prints bills for the clinic's patients would need only the PATIENT, BILLING, and PAYMENT segments. You could define the data structure shown in the following figure in a DB PCB for this program.

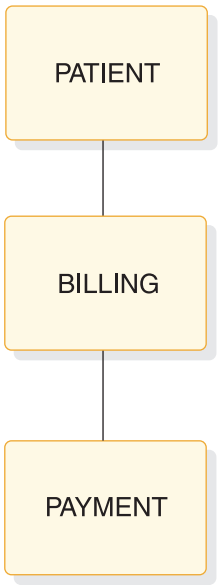


Figure 2. Accounting program's view of the database

A program that updates the database with information on patients' illnesses and treatments, in contrast, would need to process the PATIENT, ILLNESS, and TREATMNT segments. You could define the data structure shown in the following figure in a DB PCB for this program.

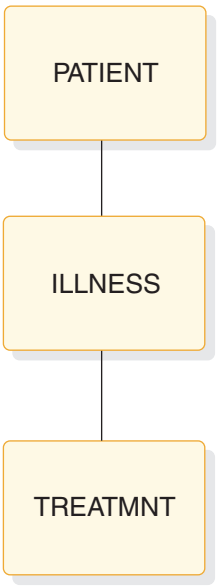


Figure 3. Patient illness program's view of the database

Sometimes a program needs to process all of the segments in the database. When this is true, the program's view of the database as defined in the DB PCB is the same as the database hierarchy that is defined in the DBD.

An application program processes only the segments in a database that it requires; therefore, if you change the format of a segment that is not processed, you do not change the program. A program is affected only by the segments that it accesses. In addition to being sensitive to only certain segments in a database, a program can also be sensitive to only certain fields within a segment. If you change a segment or field that the program is not sensitive to, it does not affect the program. You define segment and *field-level* sensitivity during PSBGEN.

Definition: *Field-level* sensitivity is when a program is sensitive to only certain fields within a segment.

Related Reading: For more information, see *IMS Version 11 Database Administration*.

Processing a database record

To process the information in the database, your application program communicates with IMS in three ways: by passing control, by communicating processing requests, and by exchanging information using DL/I calls.

- Passing control—IMS passes control to your application program through an entry statement in your program. Your program returns control to IMS when it has finished its processing.

  When you are running a CICS online program, CICS passes control to your application program, and your program schedules a PSB to make IMS requests. Your program returns control to CICS. If you are running a batch or BMP program, IMS passes control to your program with an existing PSB scheduled.
- Communicating processing requests—You communicate processing requests to IMS in one of two ways:
  - In IMS, you issue DL/I calls to process the database.
  - In CICS, you can issue either DL/I calls or EXEC DLI commands. EXEC DLI commands more closely resemble a higher-level language than do DL/I calls.

- Exchanging information using DL/I calls—Your program exchanges information in two areas:
  - A DL/I call reports the results of your request in a control block and the AIB communication block when using one of the AIB interfaces. For programs written using DL/I calls, this control block is the DB PCB. For programs written using EXEC DLI commands, this control block is the DLI interface block (DIB). The contents of the DIB reflect the status of the last DL/I command executed in the program. Your program includes a mask of the appropriate control block and uses this mask to check the results of the request.
  - When you request a segment from the database, IMS returns the segment to your I/O area. When you want to update a segment in the database, you place the new value of the segment in the I/O area.

An application program can read and update a database. When you update a database, you can replace, delete, or add segments. In IMS, you indicate in the DL/I call the segment you want to process, and whether you want to read or update it. In CICS, you can indicate what you want using either a DL/I call or an EXEC DLI command.

Tasks for developing an application

The following tasks are involved in developing an IMS application, and the programs that are part of the application.

Designing the application

Application program design varies from place to place, and from one application to another.

Therefore, this information does not try to cover the early tasks that are part of designing an application program. Instead, it covers only the tasks that you are concerned with after the early specifications for the application have been developed. The tasks for designing the application are:

- **Analyzing Application Data Requirements**

  Two important parts of application design are defining the data that each of the business processes in the application requires and designing a local view for each of the business processes.
- **Analyzing Application Processing Requirements**

  When you understand the business processes that are part of the application, you can analyze the requirements of each business process in terms of the processing that is available with different types of application programs.
- **Gathering Requirements for Database Options**

  You then need to look at the database options that can most efficiently meet the requirements, and gather information about your application's data requirements that relates to each of the options.
- **Gathering Requirements for Message Processing Options**

  If your application communicates with terminals and other application programs, look at the message processing options and the requirements they satisfy.

For more information about designing a CICS application, see *CICS Transaction Server for z/OS CICS Application Programming Guide*.

Developing specifications

Developing specifications involves defining what your application will do, and how it will be done. The task of developing specifications is not described in this information because it depends entirely on the specific application and your standards.

Implementing the design

When the specifications for each of the programs in the application are developed, you can structure and code the programs according to those specifications. The tasks of implementing the design are:

- **Writing the Database Processing Part of the Program**

  When the program design is complete, you can structure and code your requests and data areas based on the programming specifications that have been developed.
- **Writing the Message Processing Part of the Program**

  If you are writing a program that communicates with terminals and other programs, you need to structure and code the message processing part of the program.
- **Analyzing APPC/IMS Requirements**

  The LU 6.2 feature of IMS TM enables your application to be distributed throughout the network.
- **Testing an Application Program**

  When you finish coding your program, test it by itself and then as part of a system.
- **Documenting an Application Program**

  Documenting a program continues throughout the project and is most effective when done incrementally. When the program is completely tested, information must be suppled to those who use and maintain your program.

Chapter 2. Designing an application: Data and local views

Designing an application that meets the requirements of end users involves a variety of tasks and, usually, people from several departments. Application design begins when a department or business area communicates a need for some type of processing. Application design ends when each of the parts of the application system—for example, the programs, the databases, the display screens, and the message formats—have been designed.

An overview of application design

The application design process varies from place to place and from application to application. The overview that is given in this section and the suggestions about documenting application design and converting existing applications are not the only way that these tasks are performed.

The purpose of this overview is to give you a frame of reference so that you can understand where the techniques and guidelines explained in this section fit into the process. The order in which you perform the tasks described here, and the importance you give to each one, depend on your settings. Also, the individuals involved in each task, and their titles, might differ depending on the site. The tasks are as follows:

- Establish your standards

  Throughout the design process, be aware of your established standards. Some of the areas that standards are usually established for are:

  – Naming conventions (for example, for databases and terminals)
  – Formats for screens and messages
  – Control of and access to the database
  – Programming and conventions (for common routines and macros)

  Setting up standards in these areas is usually an ongoing task that is the responsibility of database and system administrators.
- Follow your security standards

  Security protects your resources from unauthorized access and use. As with defining standards, designing an adequate security system is often an ongoing task. As an application is modified or expanded, often the security must be changed in some way also. Security is an important consideration in the initial stages of application design.

  Establishing security standards and requirements is usually the responsibility of system administration. These standards are based on the requirements of your applications.

  Some security concerns are:

  – Access to and use of the databases
  – Access to terminals
  – Distribution of application output
  – Control of program modification
  – Transaction and command entry
- Define application data

© Copyright IBM Corp. 1974, 2015

Identifying the data that an application requires is a major part of application design. One of the tasks of data definition is learning from end users what information will be required to perform the required processing.

- Provide input for database design

  To design a database that meets the requirements of all the applications that will process it, the database administrator (DBA) needs information about the data requirements of each application. One way to gather and supply this information is to design a local view for each of the business processes in your application. A local view is a description of the data that a particular business process requires.

- Design application programs

  When the overall application flow and system externals have been defined, you define the programs that will perform the required processing. Some of the most important considerations involved in this task are: standards, security requirements, privacy requirements, and performance requirements. The specifications you develop for the programs should include:

  – Security requirements
  – Input and output data formats and volumes
  – Data verification and validation requirements
  – Logic specifications
  – Performance requirements
  – Recovery requirements
  – Linkage requirements and conventions
  – Data availability considerations

  In addition, you might be asked to provide some information about your application to the people responsible for network and user interface design.

- Document the application design process

  Recording information about the application design process is valuable to others who work with the application now and in the future. One kind of information that is helpful is information about why you designed the application the way you did. This information can be helpful to people who are responsible for the database, your IMS system, and the programs in the application—especially if any part of the application must be changed in the future. Documenting application design is done most thoroughly when it is done during the design process, instead of at the end of it.

- Convert an existing application

  One of the main aspects in converting an existing application to IMS is to know what already exists. Before starting to convert the existing system, find out everything you can about the way it works currently. For example, the following information can be of help to you when you begin the conversion:

  – Record layouts of all records used by the application
  – Number of data element occurrences for each data element
  – Structure of any existing related databases

Related concepts:

"Providing data security" on page 91

"Identifying online security requirements" on page 99

"Identifying application data" on page 17

"Designing a local view" on page 22

Identifying application data

Two important aspects of application design are identifying the application data and describing the data that a particular business process requires.

One of the steps of identifying application data is to thoroughly understand the processing the user wants performed. You need to understand the input data and the required output data in order to define the data requirements of the application. You also need to understand the business processes that are involved in the user's processing needs. Three of the tasks involved in identifying application data are:

- Listing the data required by the business process
- Naming the data
- Documenting the data

When analyzing the required application data, you can categorize the data as either an entity or a data element.

Definitions: An *entity* is anything about which information can be stored. A *data element* is the smallest named unit of data pertaining to an entity. It is information that describes the entity.

Example: In an education application, "students" and "courses" are both entities; these are two subjects about which you collect and process data. The following table shows some data elements that relate to the student and course entities. The entity is listed with its related data elements.

Table 10. Entities and data elements.

Entity	Data elements
Student	Student Name
	Student Number
Course	Course Name
	Course Number
	Course Length

When you store this data in an IMS database, groups of data elements are potential segments in the hierarchy. Each data element is a potential field in that segment.

Related concepts:

"An overview of application design" on page 15

Listing data elements

To identify application data, you list its data elements.

For example, to identify application data, consider a company that provides technical education to its customers. The education company has one headquarters office, called Headquarters, and several local education centers, called Ed Centers.

A class is a single offering of a course on a specific date at a particular Ed Center. One course might have several offerings at different Ed Centers; each of these is a separate class. Headquarters is responsible for developing all the courses that will be offered, and each Ed Center is responsible for scheduling classes and enrolling students for its classes.

Suppose that one of the education company's requirements is for each Ed Center to print weekly current rosters for all classes at the Ed Center. The current roster is to give information about the class and the students enrolled in the class. Headquarters wants the current rosters to be in the format shown in the following figure.

```
CHICAGO                                                    01/04/04

     TRANSISTOR THEORY                    41837
     10 DAYS
     INSTRUCTOR(S): BENSON, R.J.          DATE: 01/14/04

   STUDENT        CUST   LOCATION              STATUS   ABSENT   GRADE
 1.ADAMS, J.W.    XYZ    SOUTH BEND, IND       CONF
 2.BAKER, R.T.    ACME   BENTON HARBOR, MICH   WAIT
 3.DRAKE, R.A.    XYZ    SOUTH BEND, IND       CANC
   .
   .
   .
33.WILLIAMS, L.R. BEST   CHICAGO, ILL          CONF


  CONFIRMED = 30
  WAIT-LISTED = 1
  CANCELED = 2
```

Figure 4. Current roster for technical education example

To list the data elements for a particular business process, look at the required output. The current roster shown in the previous figure is the roster for the class, "Transistor Theory" to be given in the Chicago Ed Center, starting on January 14, 2004, for ten days. Each course has a course code associated with it—in this case, 41837. The code for a particular course is always the same. For example, if Transistor Theory is also offered in New York, the course code is still 41837. The roster also gives the names of the instructors who are teaching the course. Although the example only shows one instructor, a course might require more than one instructor.

For each student, the roster keeps the following information: a sequence number for each student, the student's name, the student's company (CUST), the company's location, the student's status in the class, and the student's absences and grade. All the above information on the course and the students is input information.

The current date (the date that the roster is printed) is displayed in the upper right corner (01/04/04). The current date is an example of data that is output only data; it is generated by the operating system and is not stored in the database.

The bottom-left corner gives a summary of the class status. This data is not included in the input data. These values are determined by the program during processing.

When you list the data elements, abbreviating them is helpful, because you will be referring to them frequently when you design the local view.

The data elements list for current roster is:

EDCNTR
: Name of Ed Center giving class

DATE Date class starts

CRSNAME
Name of course

CRSCODE
Course code

LENGTH
Length of course

INSTRS
Names of instructors teaching class

STUSEQ#
Student's sequence number

STUNAME
Student's name

CUST Name of student's company

LOCTN
Location of student's company

STATUS
Student's status in class—confirmed, wait list, or cancelled

ABSENCE
Number of days student was absent

GRADE
Student's grade for the course

After you have listed the data elements, choose the major entity that these elements describe. In this case, the major entity is class. Although a lot of information exists about each student and some information exists about the course in general, together all this information relates to a specific class. If the information about each student (for example, status, absence, and grade) is not related to a particular class, the information is meaningless. This holds true for the data elements at the top of the list as well: The Ed Center, the date the class starts, and the instructor mean nothing unless you know what class they describe.

Naming data elements

Some of the data elements your application uses might already exist and be named. After you have listed the data elements, find out if any of them exist by checking with your database administrator (DBA).

Before you begin naming data elements, be aware of the naming standards that you are subject to. When you name data elements, use the most descriptive names possible. Remember that, because other applications probably use at least some of the same data, the names should mean the same thing to everyone. Try not to limit the name's meaning only to your application.

Recommendation: Use global names rather than local names. A *global name* is a name whose meaning is clear outside of any particular application. A *local name* is a name that, to be understood, must be seen in the context of a particular application.

One of the problems with using local names is that you can develop synonyms, two names for the same data element.

For example, in the current roster example, suppose the student's company was referred to simply as “company” instead of “customer”. But suppose the accounting department for the education company used the same piece of data in a billing application—the name of the student's company—and referred to it as “customer”. This would mean that two business processes were using two different names for the same piece of data. At worst, this could lead to redundant data if no one realized that “customer” and “company” contained the same data. To solve this, use a global name that is recognized by both departments using this data element. In this case, “customer” is more easily recognized and the better choice. This name uniquely identifies the data element and has a specific meaning within the education company.

When you choose data element names, use qualifiers so that each name can mean only one thing.

For example, suppose Headquarters, for each course that is taught, assigns a number to the course as it is developed and calls this number the “sequence number”. The Ed Centers, as they receive student enrollments for a particular class, assign a number to each student as a means of identification within the class. The Ed Centers call **this** number the “sequence number”. Thus Headquarters and the Ed Centers are using the same name for two separate data elements. This is called a *homonym*. You can solve the homonym problem by qualifying the names. The number that Headquarters assigns to each course can be called “course code” (CRSCODE), and the number that the Ed Centers assign to their students can be called “student sequence number” (STUSEQ#).

Homonym
One word for two different things.

Choose data element names that identify the element and describe it precisely. Make your data element names:

Unique
The name is clearly distinguishable from other names.

Self-explanatory
The name is easily understood and recognized.

Concise
The name is descriptive in a few words.

Universal
The name means the same thing to everyone.

Documenting application data

After you have determined what data elements a business process requires, record as much information about each of the data elements as possible.

This information is useful to the DBA. Be aware of any standards that you are subject to regarding data documentation. Many places have standards concerning what information should be recorded about data and how and where that information should be recorded. The amount and type of this information varies from place to place. The following list is the type of information that is often recorded.

The descriptive name of the data element
Data element names should be precise, yet they should be meaningful to people who are familiar and also to those who are unfamiliar with the application.

The length of the data element
The length of the data element determines segment size and segment format.

The character format
The programmer needs to know if the data is alphanumeric, hexadecimal, packed decimal, or binary.

The range of possible values for the element
The range of possible values for the element is important for validity checking.

The default value
The programmer also needs the default value.

The number of data element occurrences
The number of data element occurrences helps the DBA to determine the required space for this data, and it affects performance considerations.

How the business process affects the data element
Whether the data element is read or updated determines the processing option that is coded in the PSB for the application program.

You should also record control information about the data. Such information should address the following questions:

- What action should the program take when the data it attempts to access is not available?
- If the format of a particular data element changes, which business processes does that affect? For example, if an education database has as one of its data elements a five-digit code for each course, and the code is changed to six digits, which business processes does this affect?
- Where is the data now? Know the sources of the data elements required by the application.
- Which business processes make changes to a particular data element?
- Are there security requirements about the data in your application? For example, you would not want information such as employees' salaries available to everyone?
- Which department owns and controls the data?

One way to gather and record this information is to use a form similar to the one shown in the following table. The amount and type of data that you record depends on the standards that you are subject to. For example, the following table lists the ID number, data element name, length, the character format, the allowed, null, default values, and the number of occurrences.

Table 11. Example of data elements information form

ID #	Data element name	Length	Char. format	Allowed values	Null values	Default value	Number of occurrences
5	Course Code	5 bytes	Hexa-decimal	0010090000	00000	N/A	There are 200 courses in the curriculum. An average of 10 are new or revised per year. An average of 5 are dropped per year.
25	Status	4 bytes	Alpha-numeric	CONF WAIT CANC	blanks	WAIT	1 per student
36	Student Name	20 bytes	Alpha-numeric	Alpha only	blanks	N/A	There are 3 to 100 students per class with an average of 40 per class.

A *data dictionary* is a good place to record the facts about the application's data. When you are analyzing data, a dictionary can help you find out whether a particular data element already exists, and if it does, its characteristics. With the IBM OS/VS DB/DC Data Dictionary, you can determine online what segments exist in a particular database and what fields those segments contain. You can use either tool to create reports involving the same information.

Designing a local view

A *local view* is a description of the data that an individual business process requires.

It includes the following:

- A list of the data elements
- A conceptual data structure that shows how you have grouped data elements by the entities that they describe
- The relationships between each of the groups of data elements

Definitions: A *data aggregate* is a group of data elements. When you have grouped data elements by the entity they describe, you can determine the relationships between the data aggregates. These relationships are called *mappings*. Based on the mappings, you can design a conceptual data structure for the business process. You should document this process as well.

Related concepts:

"An overview of application design" on page 15

Analyzing data relationships

When you analyze data relationships, you are developing conceptual data structures for the business processes in your application.

This process, called *data structuring*, is a way to analyze the relationships among the data elements a business process requires, not a way to design a database. The decisions about segment formats and contents belong to the DBA. The information you develop is input for designing a database.

Data structuring can be done in many different ways.

Grouping data elements into hierarchies

The data elements that describe a data aggregate, the student, might be represented by the descriptive names STUSEQ#, STUNAME, CUST, LOCTN, STATUS, ABSENCE, and GRADE. We call this group of data elements the student data aggregate.

Data elements have values and names. In the student data elements example, the values are a particular student's sequence number, the student's name, company, company location, the student's status in the class, the student's absences, and grade. The names of the data aggregate are not unique—they describe all the students in the class in the same terms. The combined values, however, of a data aggregate occurrence are unique. No two students can have the same values in each of these fields.

As you group data elements into data aggregates and data structures, look at the data elements that make up each group and choose one or more data elements that uniquely identify that group. This is the data aggregate's *controlling key*, which is the data element or group of data elements in the aggregate that uniquely identifies the aggregate. Sometimes you must use more than one data element for the key in order to uniquely identify the aggregate.

By following the three steps explained in this section, you can develop a conceptual data structure for a business process's data. However, you are not developing the logical data structure for the program that performs the business process. The three steps are:

1. Separate repeating data elements in a single occurrence of the data aggregate.
2. Separate duplicate values in multiple occurrences of the data aggregate.
3. Group each data element with its controlling keys.

Step 1. separating repeating data elements

Look at a single occurrence of the data aggregate. The following table shows what this looks like for the class aggregate; the data element is listed with the class aggregate occurrence.

Table 12. Single occurrence of class aggregate

Data element	Class aggregate occurrence
EDCNTR	CHICAGO
DATE(START)	1/14/96
CRSNAME	TRANSISTOR THEORY
CRS CODE	41837
LENGTH	10 DAYS
INSTRS	multiple
STUSEQ#	multiple
STUNAME	multiple
CUST	multiple
LOCTN	multiple
STATUS	multiple
ABSENCE	multiple
GRADE	multiple

The data elements defined as multiple are the elements that repeat. Separate the repeating data elements by shifting them to a lower level. Keep data elements with their controlling keys.

The data elements that repeat for a single class are: STUSEQ#, STUNAME, CUST, LOCTN, STATUS, ABSENCE, and GRADE. INSTRS is also a repeating data element, because some classes require two instructors, although this class requires only one.

When you separate repeating data elements into groups, you have the structure shown in the following figure.

In the following figure, the data elements in each box form an aggregate. The entire figure depicts a data structure. The data elements include the Course aggregate, the Student aggregate, and the Instructor aggregate.

The following figure shows these aggregates with the keys indicated with leading asterisks (*).

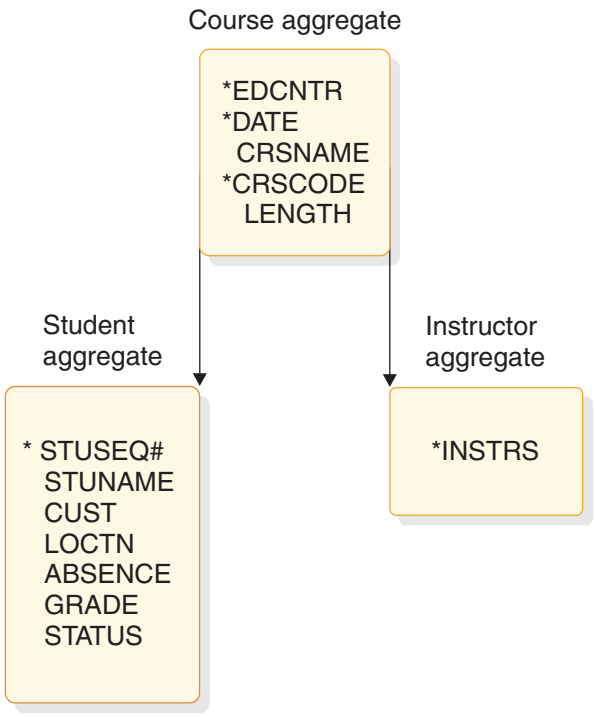


Figure 5. Current roster after step 1

The keys for the data aggregates are shown in the following table.

Table 13. Data aggregates and keys for current roster after step 1

Data aggregate	Keys
Course aggregate	EDCNTR, DATE, CRSCODE
Student aggregate	EDCNTR, DATE, CRSCODE, STUSEQ#
Instructor aggregate	EDCNTR, DATE, CRSCODE, INSTRS

The asterisks in the previous figure identify the key data elements. For the Class aggregate, it takes multiple data elements to identify the course, so you need multiple data elements to make up the key. The data elements that comprise the Class aggregate are:

- Controlling key element, STUSEQ#
- STUNAME
- CUST
- LOCTN
- STATUS
- ABSENCE
- GRADE

The data elements that comprise the Instructor aggregate are:

- Key element, INSTRS

The Course aggregate and the Instructor aggregate inherit the following keys from the root segment, Course aggregate:

- EDCNTR
- DATE
- CRSCODE

After you have shifted repeating data elements, make sure that each element is in the same group as its controlling key. INSTRS is separated from the group of data elements describing a student because the information about instructors is unrelated to the information about the students. The student sequence number does not control who the instructor is.

In the example shown in the previous figure, the Student aggregate and Instructor aggregate are both dependents of the Course aggregate. A dependent aggregate's key includes the concatenated keys of all the aggregates above the dependent aggregate. This is because a dependent's controlling key does not mean anything if you do not know the keys of the higher aggregates. For example, if you knew that a student's sequence number was 4, you would be able to find out all the information about the student associated with that number. This number would be meaningless, however, if it were not associated with a particular course. But, because the key for the Student aggregate is made up of Ed Center, date, and course code, you can deduce which class the student is in.

Step 2. isolating duplicate aggregate values

Look at multiple occurrences of the aggregate—in this case, the values you might have for two classes. The following table shows multiple occurrences (2) of the same data elements. As you look at this table, check for duplicate values. Remember that both occurrences describe one course.

Table 14. Multiple occurrences of class aggregate

Data element list	Occurrence 1	Occurrence 2
EDCNTR	CHICAGO	NEW YORK
DATE(START)	1/14/96	3/10/96
CRSNAME	TRANS THEORY	TRANS THEORY
CRSCODE	41837	41837

Table 14. Multiple occurrences of class aggregate (continued)

Data element list	Occurrence 1	Occurrence 2
LENGTH	10 DAYS	10 DAYS
INSTRS	multiple	multiple
STUSEQ#	multiple	multiple
STUNAME	multiple	multiple
CUST	multiple	multiple
LOCTN	multiple	multiple
STATUS	multiple	multiple
ABSENCE	multiple	multiple
GRADE	multiple	multiple

The data elements defined as multiple are the data elements that repeat. The values in these elements are not the same. The aggregate is always unique for a particular class.

In this step, compare the two occurrences and shift the fields with duplicate values (TRANS THEORY and so on) to a higher level. If you need to, choose a controlling key for aggregates that do not yet have keys.

In the previous table, CRSNAME, CRSCODE, and LENGTH are the fields that have duplicate values. Much of this process is intuitive. Student status and grade, although they can have duplicate values, should not be separated because they are not meaningful values by themselves. These values would not be used to identify a particular student. This becomes clear when you remember to keep data elements with their controlling keys. When you separate duplicate values, you have the structure shown in the following figure.

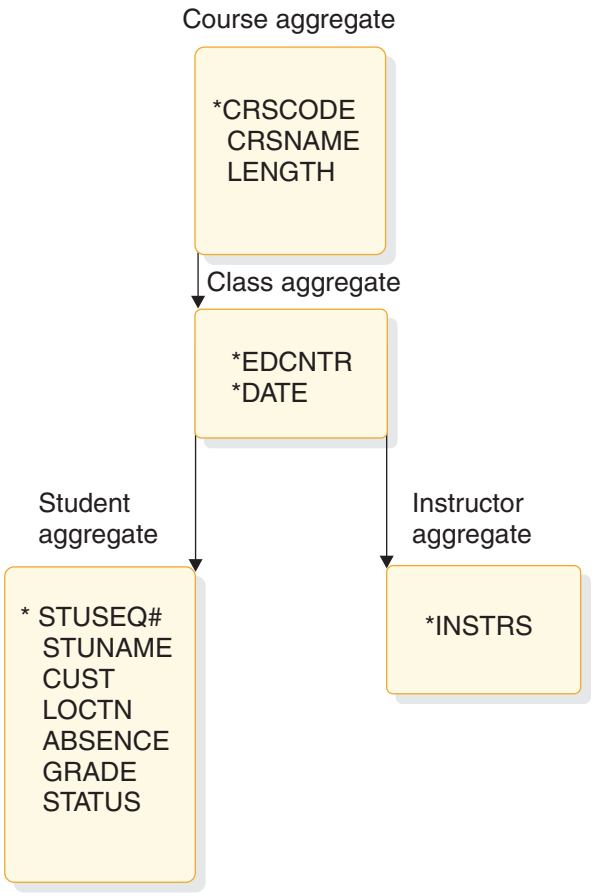


Figure 6. Current roster after step 2

Step 3. grouping data elements with their controlling keys

This step is often a check on the first two steps. (Sometimes the first two steps have already done what this step instructs you to do.)

At this stage, make sure that each data element is in the group that contains its controlling key. The data element should depend on the full key. If the data element depends only on part of the key, separate the data element along with the partial (controlling) key on which it depends.

In this example, CUST and LOCTN do not depend on the STUSEQ#. They are related to the student, but they do not depend on the student. They identify the company and company address of the student.

CUST and LOCTN are not dependent on the course, the Ed Center, or the date, either. They are separate from all of these things. Because a student is only associated with one CUST and LOCTN, but a CUST and LOCTN can have many students attending classes, the CUST and LOCTN aggregate should be above the student aggregate.

The following figure shows these aggregates and keys indicated with leading asterisks (*) and shows what the structure looks like when you separate CUST and LOCTN.

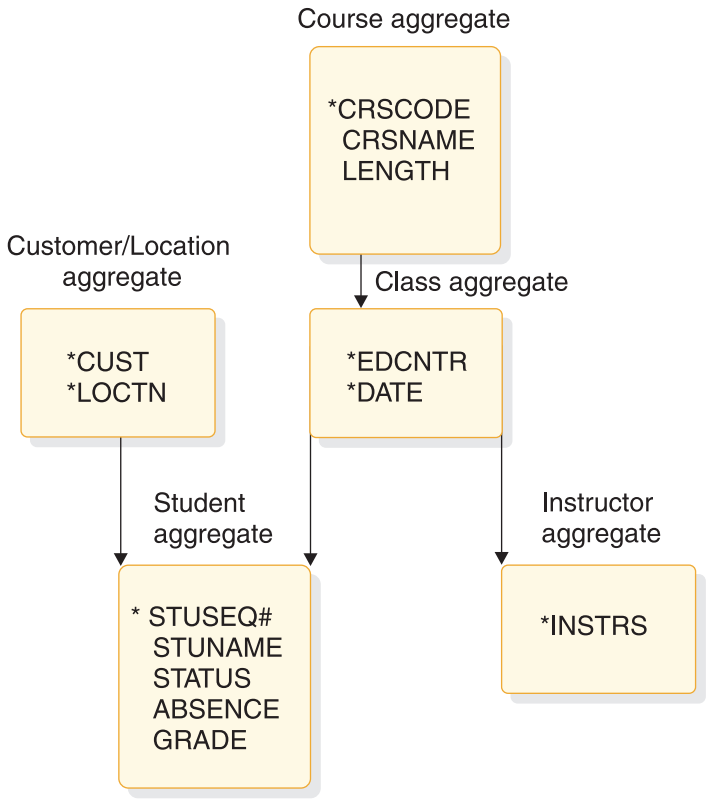


Figure 7. Current roster after step 3

The keys for the data aggregates are shown in the following table.

Table 15. Data aggregates and keys for current roster after step 3

Data aggregate	Keys
Course aggregate	CRSCODE
Class aggregate	CRSCODE, EDCNTR, DATE
Customer aggregate	CUST, LOCTN
Student aggregate	(when viewed from the customer aggregate in "Current roster after step 3"instead of from the course aggregate, in "Current roster after step 2") CUST, LOCTN, STUSEQ, CRSCODE, EDCNTR, DATE
Instructor aggregate	CRSCODE, EDCNTR, DATE, INSTRS

Deciding on the arrangement of the customer and location information is part of designing a database. Data structuring should separate any inconsistent data elements from the rest of the data elements.

Determining mappings

When you have arranged the data aggregates into a conceptual data structure, you can examine the relationships between the data aggregates. A mapping between two data aggregates is the quantitative relationship between the two.

The reason you record mappings is that they reflect relationships between segments in the data structure that you have developed. If you store this information in an IMS database, the DBA can construct a database hierarchy that

satisfies all the local views, based on the mappings. In determining mappings, it is easier to refer to the data aggregates by their keys, rather than by their collected data elements.

The two possible relationships between any two data aggregates are:

- One-to-many

  For each segment A, one or more occurrences of segment B exist. For example, each class maps to one or more students.

  Mapping notation shows this in the following way:

  Class ◄——►► Student

- Many-to-many

  Segment B has many A segments associated with it and segment A has many B segments associated with it. In a hierarchic data structure, a parent can have one or more children, but each child can be associated with only one parent. The many-to-many association does not fit into a hierarchy, because in a many-to-many association each child can be associated with more than one parent.

  Related Reading: For more information about analyzing data requirements, see *IMS Version 11 Database Administration*.

  Many-to-many relationships occur between segments in two business processes. A many-to-many relationship indicates a conflict in the way that two business processes need to process those data aggregates. If you use the IMS full-function database, you can solve this kind of processing conflict by using secondary indexing or logical relationships.

The mappings for the current roster are:

- **Course ◄——►► Class**

  For each course, there might be several classes scheduled, but a class is associated with only one course.

- **Class ◄——►► Student**

  A class has many students enrolled in it, but a student might be in only one class offering of this course.

- **Class ◄——►► Instructor**

  A class might have more than one instructor, but an instructor only teaches one class at a time.

- **Customer/location ◄——►► Student**

  A customer might have several students attending a particular class, but each student is only associated with one customer and location.

Related concepts:

"Understanding how data structure conflicts are resolved" on page 81

Local view examples

The following examples show how to design local views including the schedule of courses, the instructor skills report, and the instructor schedules.

Each example shows the following parts of designing a local view:

1. Gather the data. For each example, the data elements are listed and two occurrences of the data aggregate are shown. Two occurrences are shown because you need to look at both occurrences when you look for repeating fields and duplicate values.

2. Analyze the data relationships. First, group the data elements into a conceptual data structure using these three steps:
   a. Separate repeating data elements in a single occurrence of the data aggregate by shifting them to a lower level. Keep data elements with their keys.
   b. Separate duplicating values in two occurrences of the data aggregate by shifting those data elements to a higher level. Again, keep data elements with their keys.
   c. Group data elements with their keys. Make sure that all the data elements within one aggregate have the same key. Separate any that do not.
3. Determine the mappings between the data aggregates in the data structure you have developed.

Example 1: schedule of courses

Headquarters keeps a schedule of all the courses given each quarter and distributes it monthly. Headquarters wants the schedule to be sorted by course code and printed in the format shown in the following figure.

```
           COURSE SCHEDULE

COURSE:   TRANSISTOR THEORY       COURSE CODE:    418737
LENGTH:   10 DAYS                 PRICE:          $280

DATE                      LOCATION

APRIL 14                  BOSTON
APIRL 21                  CHICAGO
.
.
.
NOVEMBER 18               LOS ANGELES
```

Figure 8. Schedule of courses

1. Gather the data. The following table lists the data elements and two occurrences of the data aggregate.

Table 16. Course schedule data elements

Data elements	Occurrence 1	Occurrence 2
CRSNAME	TRANS THEORY	MICRO PROG
CRSCODE	41837	41840
LENGTH	10 DAYS	5 DAYS
PRICE	$280	$150
DATE	multiple	multiple
EDCNTR	multiple	multiple

2. Analyze the data relationships. First, group the data elements into a conceptual data structure.
   a. Separate repeating data elements in one occurrence of the data aggregate by shifting them to a lower level, as shown in the following table

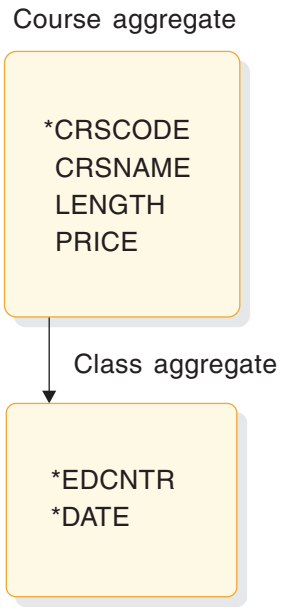


Figure 9. Course schedule after step 1

b. Next, separate duplicate values in two occurrences of the data aggregate by shifting the data elements to a higher level.

   This data aggregate does not contain duplicate values.

c. Group data elements with their controlling keys.

   Data elements are grouped with their keys in the present structure. No changes are necessary for this step.

   The keys for the data aggregates are shown in the following table.

Table 17. Data aggregates and keys for course schedule after step 1

Data aggregate	Keys
Course aggregate	CRSCODE
Class aggregate	CRSCODE, EDCNTR, DATE

3. When you have developed a conceptual data structure, determine the mappings for the data aggregates.

   The mapping for this local view is: **Course** ◄——►► **Class**

Example 2: instructor skills report

Each Ed Center needs to print a report showing the courses that its instructors are qualified to teach. The report format is shown in the following figure.

```
                    INSTRUCTOR SKILLS REPORT

INSTRUCTOR          COURSE CODE         COURSE NAME

BENSON, R. J.       41837               TRANS THEORY
MORRIS, S. R.       41837               TRANS THEORY
                    41850               CIRCUIT DESIGN
                    41852               LOGIC THEORY
.
.
.
REYNOLDS, P. W.     41840               MICRO PROG
                    41850               CIRCUIT DESIGN
```

Figure 10. Instructor skills report

1. Gather the data. The following table lists the data elements and two occurrences of the data aggregate.

Table 18. Instructor skills data elements

Data elements	Occurrence 1	Occurrence 2
INSTR	REYNOLDS, P.W.	MORRIS, S. R.
CRSCODE	multiple	multiple
CRSNAME	multiple	multiple

2. Analyze the data relationships. First, group the data elements into a conceptual data structure.
   a. Separate repeating data elements in one occurrence of the data aggregate by shifting to a higher level as shown in the following figure.

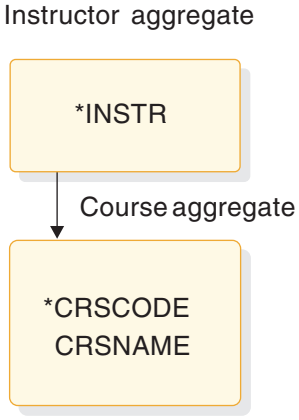


Figure 11. Instructor skills after step 1

   b. Separate any duplicate values in the two occurrences of the data aggregate.
      No duplicate values exist in this data aggregate.
   c. Group data elements with their keys.
      All data elements are grouped with their keys in the current data structure. There are no changes to this data structure.
3. Determine the mappings for the data aggregates.
   The mapping for this local view is: **Instructor** ◄——►► **Course**

Example 3: instructor schedules

Headquarters wants to produce a report showing the schedules for all the instructors. The following figure shows the report format.

```
                        INSTRUCTOR SCHEDULES

INSTRUCTOR          COURSE           CODE     ED CENTER     DATE

BENSON, R. J.       TRANS THEORY     41837    CHICAGO       1/14/96
MORRIS, S. R.       TRANS THEORY     41837    NEW YORK      3/10/96
                    LOGIC THEORY     41852    BOSTON        3/27/96
                    CIRCUIT DES      41840    CHICAGO       4/21/96
REYNOLDS, B. H.     MICRO PROG       41850    NEW YORK      2/25/96
                    CIRCUIT DES      41850    LOS ANGELES   3/10.96
```

Figure 12. Instructor schedules

1. Gather the data. The following table lists the data elements and two occurrences of the data aggregate.

Table 19. Instructor schedules data elements

Data elements	Occurrence 1	Occurrence 2
INSTR	BENSON, R. J.	MORRIS, S. R.
CRSNAME	multiple	multiple
CRSCODE	multiple	multiple
EDCNTR	multiple	multiple
DATE(START)	multiple	multiple

2. Analyze the data relationships. First, group the data elements into a conceptual data structure.
   a. Separate repeating data elements in one occurrence of the data aggregate by shifting data elements to a lower level as shown in the following figure.

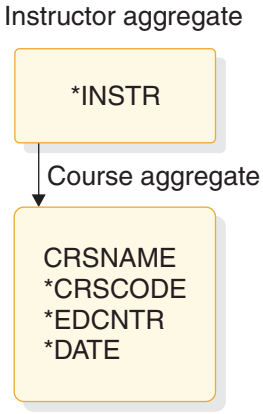


Figure 13. Instructor schedules step 1

   b. Separate duplicate values in two occurrences of the data aggregate by shifting data elements to a higher level as shown in the following figure.

      In this example, CRSNAME and CRSCODE can be duplicated for one instructor or for many instructors, for example, 41837 for Benson and 41850 for Morris and Reynolds.

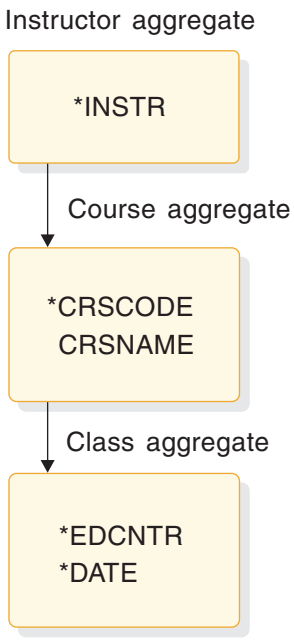


Figure 14. Instructor schedules step 2

c. Group data elements with their keys.

   All data elements are grouped with their controlling keys in the current data structure. No changes to the current data structure are required.

3. Determine the mappings for the data aggregates.

   The mappings for this local view are: **Instructor** ◄——►► **Course Course** ◄——►► **Class**

   An analysis of data requirements is necessary to combine the requirements of the three examples presented in this section and to design a hierarchic structure for the database based on these requirements.

   Related Reading: For more information on analyzing data requirements, see *IMS Version 11 Database Administration*.

Chapter 3. Analyzing IMS application processing requirements

Use the following information to plan for writing application programs for IMS environments.

Defining IMS application requirements

One of the steps of application design is to decide how the business processes, or tasks, that the end user wants performed can be best grouped into a set of programs that efficiently performs the required processing.

To analyze processing requirements, consider:

- **When the task must be performed**
  - Will the task be scheduled unpredictably (for example, on terminal demand) or periodically (for example, weekly)?
- **How the program that performs the task is executed**
  - Will the program be executed online, where response time is crucial, or by batch job submission, where a slower response time is acceptable?
- **The consistency of the processing components**
  - Does the action the program is to perform involve more than one type of program logic? For example, does it involve mostly retrievals and only one or two updates? If so, you should consider separating the updates into a separate program.
  - Does this action involve several large groups of data? If it does, it might be more efficient to separate the programs by the data they access.
- **Any special requirements about the data or processing**

  Security
  Should access to the program be restricted?

  Recovery
  Are there special recovery considerations in the program's processing?

  Availability
  Does your application require high data availability?

  Integrity
  Do other departments use the same data?

Answers to questions like these can help you decide on the number of application programs that the processing will require, and on the types of programs that perform the processing most efficiently. Although rules dealing with how many programs can most efficiently do the required processing do not exist, here are some suggestions:

- As you look at each programming task, examine the data and processing that each task involves. If a task requires different types of processing and has different time limitations (for example, daily as opposed to different times throughout the month), that task might be more efficiently performed by several programs.
- As you define each program, it is a good idea for maintenance and recovery reasons to keep it as simple as possible. The simpler a program is—the less it does—the easier it is to maintain, and to restart after a program or system

© Copyright IBM Corp. 1974, 2015

failure. The same is true with data availability—the less data that is accessed, the more likely the data is to be available. The more limited the access requested, the more likely the data is to be available.

Similarly, if the data that the application requires is physically in one place, it might be more efficient to have one program do more of the processing than usual. These are considerations that depend upon the processing and the data of each application.

- Documenting each of the user tasks is helpful during the design process, and in the future when others will work with your application. Be sure you are aware of standards in this area. The kind of information that is typically kept is when the action is to be executed, a functional description, and requirements for maintenance, security, and recovery.

  For example, for the current roster process described previously, you might record the information shown in the following form. How frequently the program is run is determined by the number of classes (20) needed by the Education Center each week.

Documenting user task descriptions: current roster example

```
USER TASK DESCRIPTION

NAME: Current Roster
ENVIRONMENT: Batch                    FREQUENCY: 20 per week

INVOKING EVENT OR DOCUMENT: Time period (one week)

REQUIRED RESPONSE TIME: 24 hours

FUNCTION DESCRIPTION: Print weekly, a current student roster, in student
 number sequence for each class offered at the Education Center.

MAINTENANCE:Included in Education DB maintenance.

SECURITY: None.

RECOVERY:After a failure, the ability to start printing a particular
class roster starting from a particular sequential student number.
```

Accessing databases with your IMS application program

When designing your program, consider the type of database it must access. The type of database depends on the operating environment.

The program types you can run and the different types of databases you can access in a DB batch, TM batch, DB/DC, DBCTL, or DCCTL environment are shown in the following table.

Table 20. Program and database options in IMS environments

Environment	Type of program you can run	Type of database that can be accessed
DB/DC	BMP	DB2® for z/OS DEDB and MSDB Full function z/OS files
	IFP	DB2 for z/OS DEDB Full function
	JBP	DB2 for z/OS DEDB Full function
	JMP	DB2 for z/OS DEDB Full function
	MPP	DB2 for z/OS DEDB and MSDB Full function
DB Batch	DB Batch	DB2 for z/OS Full function GSAM z/OS files
DBCTL	BMP (Batch-oriented)	DB2 for z/OS DEDB Full function GSAM z/OS files
	JBP	DB2 for z/OS DEDB Full function
DCCTL	BMP	DB2 for z/OS GSAM z/OS files
	IFP	DB2 for z/OS
	JMP	DB2 for z/OS
	MPP	DB2 for z/OS
TM Batch	TM Batch	DB2 for z/OS GSAM z/OS files

The types of databases that can be accessed are:

- **IMS Databases**

There are two types of IMS databases: full-function and Fast Path.

– **Full-function databases**

  Full-function databases are hierarchic databases that are accessed through Data Language I (DL/I) call interface and can be processed by these types of application programs: IFP, JMP, JBP, MPP, BMP, and DB batch. DL/I calls make it possible for IMS application programs to retrieve, replace, delete, and add segments to full-function databases.

  JMP and JBP applications use JDBC to access full-function databases in addition to DL/I.

  If you use data sharing, online programs and batch programs can access the same full-function database concurrently.

  Full-function database types include: HDAM, HIDAM, HSAM, HISAM, PHDAM, PHIDAM, SHSAM, and SHISAM.

– **Fast Path databases**

  Fast Path databases are of two types: MSDBs and DEDBs.

  - **Main storage databases** (MSDBs) are root-segment-only databases that reside in virtual storage during execution.
  - **Data entry databases** (DEDBs) are hierarchic databases that provide a high level of availability for, and efficient access to, large volumes of detailed data.

  MPP, BMP, and IFP programs can access Fast Path databases. In the DBCTL environment, BMP programs can access DEDBs but not MSDBs. JMP and JBP programs can access DEDBs but not MSDBs.

- **DB2 for z/OS databases**

  DB2 for z/OS databases are relational databases that can be processed by IMS batch, BMP, IFP, JBP, JMP, and MPP programs. An IMS application program might access only DL/I databases, both DL/I and DB2 for z/OS databases, or only DB2 for z/OS databases. Relational databases are represented to application programs and users as tables, and are processed using a relational data language called Structured Query Language (SQL).

  Note: JMP and JBP programs cannot access DB2 for z/OS databases.

  Related Reading: For information on processing DB2 for z/OS databases, see DB2 for z/OS Application Programming and SQL Guide.

- **z/OS Files**

  BMPs (in DB/DC, DBCTL, and DCCTL environments) are the only type of online application program that can access z/OS files for their input or output. Batch programs can also access z/OS files.

- **GSAM Databases** (Generalized Sequential Access Method)

  Generalized Sequential Access Method (GSAM) is an access method that makes it possible for BMPs and batch programs to access a sequential z/OS data set as a simple database. A GSAM database can be accessed by z/OS or by IMS.

Accessing data: the types of programs you can write for your IMS application

You must decide what type of program to use: batch programs, message processing programs (MPPs), IMS Fast Path (IFP) applications, batch message processing (BMP) applications, Java Message Processing (JMP) applications, or Java Batch Processing (JBP) applications. The types of programs you can use depend on whether you are running in the batch, DB/DC, or DBCTL environment.

DB batch processing

These topics describe DB batch processing and can help you decide if this batch program is appropriate for your application.

Data that a DB batch program can access

A DB batch program can access full-function databases, DB2 for z/OS databases, GSAM databases, and z/OS files. A DB batch program cannot access DEDBs or MSDBs.

Using DB batch processing

Batch programs are typically longer-running programs than online programs. You use a batch program when you have a large number of database updates to do or a report to print. Because a batch program runs by itself—it does not compete with any other programs for resources like databases—it can run independently of the control region. If you use data sharing, DB batch programs and online programs can access full-function databases concurrently. Batch programs:

- Typically produce a large amount of output, such as reports.
- Are not executed by another program or user. They are usually scheduled at specific time intervals (for example, weekly) and are started with JCL.
- Produce output that is not needed right away. The turnaround time for batch output is not crucial, as it usually is for online programs.

Recovering a DB batch program

Include checkpoints in your batch program to restart it in case of failure.

Issuing checkpoints

Issue checkpoints in a batch program to commit database changes and provide places from which to restart your program. Issuing checkpoints in a batch program is important, because commit points do not occur automatically, as they do in MPPs, transaction-oriented BMPs, and IFPs.

Issuing checkpoints is particularly important in a batch program that participates in data sharing with your online system. Checkpoints free up resources for use by online programs. You should initially include checkpoints in all batch programs that you write. Even though the checkpoint support might not be needed then, it is easier to incorporate checkpoints initially than to try to fit them in later. And it is possible that you might want to convert your batch program to a BMP or participate in data sharing.

To issue checkpoints (or other system service calls), you must specify an I/O PCB for your program. To obtain an I/O PCB, use the compatibility option by specifying CMPAT=YES in the PSBGEN statement in your program's PSB.

Recommendation: For PSBs used by DB batch programs, always specify CMPAT=YES.

Backing out database changes

The type of storage medium for the system log determines what happens when a DB batch program terminates abnormally. You can specify that the system log be stored on either DASD (direct access storage device) or tape.

System log on DASD

If the system log is stored on DASD, using the BKO execution parameter you can specify that IMS is to dynamically back out the changes that the program has made to the database since its last commit point.

Related Reading: For information on using the BKO execution parameter, see *IMS Version 11 System Definition*.

Dynamically backing out database changes has the following advantages:

- Data accessed by the program that failed is available to other programs immediately. If batch backout is used, other programs cannot access the data until the IMS Batch Backout utility has been run to back out the database changes.
- If data sharing is being used and two programs are deadlocked, one of the programs can continue processing. Otherwise, if batch backout is used, both programs fail.

IMS performs dynamic backout for a batch program when an IMS-detected failure occurs, for example, when a deadlock is detected. Logging to DASD makes it possible for batch programs to issue the `SETS`, `ROLB`, and `ROLS` system service calls. These calls cause IMS to dynamically back out changes that the program has made.

Related Reading: For information on the `SETS`, `ROLB`, and `ROLS` calls, see the information about recovering databases and maintaining database integrity in *IMS Version 11 Database Administration*.

System log on tape

If a batch application program terminates abnormally and the batch system log is stored on tape, you must use the IMS Batch Backout utility to back out the program's changes to the database.

Related concepts:

"When to use checkpoint calls" on page 50

TM batch processing

A TM batch program acts like a DB batch program with the following differences.

- It cannot access full-function databases, but it can access DB2 for z/OS databases, GSAM databases, and z/OS files.
- To issue checkpoints for recovery, you need not specify CMPAT=YES in your program's PSB. (The CMPAT parameter is ignored in TM batch.) The I/O PCB is always the first PCB in the list.
- You cannot dynamically back out a database because IMS does not own the databases.

The IEFRDER log DD statement is required in order to enable log synchronization with other external subsystems, such as DB2 for z/OS.

Processing messages: Message Processing Programs

A Message Processing Program (MPP) is an online program that can access full-function databases, DEDBs, MSDBs, and DB2 for z/OS databases. Unlike BMPs and batch programs, MPPs cannot access GSAM databases. MPPs can only run in DB/DC and DCCTL environments.

Using an MPP

The primary purpose of an MPP is to process requests from users at terminals and from other application programs. Ideally, MPPs are very small, and the processing they perform is tailored to respond to requests quickly. They process messages as their input, and send messages as responses.

Message
Data that is transmitted between any two terminals, application programs, or IMS systems. Each message has one or more segments.

MPPs are executed through transaction codes. When you define an MPP, you associate it with one or more transaction codes. Each transaction code represents a transaction the MPP is to process. To process a transaction, a user at a terminal enters a code for that transaction. IMS then schedules the MPP associated with that code, and the MPP processes the transaction. The MPP might need to access the database to do this. Generally, an MPP goes through these five steps to process a transaction:

1. Retrieve a message from IMS.
2. Process the message and access the database as necessary.
3. Respond to the message.
4. Repeat the process until no messages are forthcoming.
5. Terminate.

When an MPP is defined, a system administrator makes decisions about the program's scheduling and processing. For each MPP, a system administrator specifies:

- The transaction's priority
- The number of messages for a particular transaction code that the MPP can process in a single scheduling
- The amount of time (in seconds) in which the MPP is allowed to process a single transaction

Defining priorities and processing limits gives system administration some control over load balancing and processing.

Although the primary purpose of an MPP is to process and reply to messages quickly, it is flexible in how it processes a transaction and where it can send output messages. For example, an MPP can send output messages to other terminals and application programs.

Related concepts:

Chapter 5, “Gathering requirements for database options,” on page 73

Processing messages: IMS Fast Path Programs

An IMS Fast Path Program (IFP) is similar to an MPP: Its main purpose is to quickly process and reply to messages from terminals. Like an MPP, an IFP can access full-function databases, DEDBs, MSDBs, and DB2 for z/OS databases. IFPs can only be run in DB/DC and DCCTL environments.

Using an IFP

You should use an IFP if you need quick processing and can accept the characteristics and constraints associated with IFPs.

The main differences between IFPs and MPPs are as follows:

- Messages processed by IFPs must consist of only one segment. Messages that are processed by MPPs can consist of several segments.
- IFPs bypass IMS queuing, allowing for more efficient processing. Transactions that are processed by Fast Path's EMH (expedited message handler) are on a first-in, first-out basis.

IFPs also have the following characteristics:

- They run in **transaction response mode**. This means that they must respond to the terminal that sent the message before the terminal can enter any more requests.
- They process only **wait-for-input transactions**. When you define a program as processing wait-for-input transactions, the program remains in virtual storage, even when no additional messages are available for it to process.

Restrictions:

- An IMS program cannot send messages to an IFP transaction unless it is in another IMS system that is connected using Intersystem Communication (ISC).
- MPPs cannot pass conversations to an IFP transaction.

Recovering an IFP

IFPs must be defined as single mode. This means that a commit point occurs each time the program retrieves a message. Because of this, you do not need to issue checkpoint calls.

Batch message processing: BMPs

BMPs are application programs that can perform batch-type processing online and access the IMS message queues for their input and output. Because of this and because of the data available to them, BMPs are the most flexible of the IMS application programs. The two types of BMPs are: batch-oriented and transaction-oriented.

Batch processing online: batch-oriented BMPs

A batch-oriented BMP performs batch-type processing in any online environment. When run in the DB/DC or DCCTL environment, a batch-oriented BMP can send its output to the IMS message queue to be processed later by another application program. Unlike a transaction-oriented BMP, a batch-oriented BMP cannot access the IMS message queue for input.

Data a batch-oriented BMP can access

In the DBCTL environment, a batch-oriented BMP can access full-function databases, DB2 for z/OS databases, DEDBs, z/OS files, and GSAM databases. In the DB/DC environment, a batch-oriented BMP can access all of these types of databases, as well as Fast Path MSDBs. In the DCCTL environment, this program can access DB2 for z/OS databases, z/OS files, and GSAM databases.

Using a batch-oriented BMP

A batch-oriented BMP can be simply a batch program that runs online. (Online requests are processed by the IMS DB/DC, DBCTL, or DCCTL system rather than by a batch system.) You can even run the same program as a BMP or as a batch program.

Recommendation: If the program performs a large number of database updates without issuing checkpoints, consider running it as a batch program so that it does not degrade the performance of the online system.

To use batch-oriented BMPs most efficiently, avoid a large amount of batch-type processing online. If you have a BMP that performs time-consuming processing such as report writing and database scanning, schedule it during non-peak hours of processing. This will prevent it from degrading the response time of MPPs.

Because BMPs can degrade response times, your response time requirements should be the main consideration in deciding the extent to which you will use batch message processing. Therefore, use BMPs accordingly.

Recovering a batch-oriented BMP

Issuing checkpoint calls is an important part of batch-oriented BMP processing, because commit points do not occur automatically, as they do in MPPs, transaction-oriented BMPs, and IFPs. Unlike most batch programs, a BMP shares resources with MPPs. In addition to committing database changes and providing places from which to restart (as for a batch program), checkpoints release resources that are locked for the program.

If a batch-oriented BMP fails, IMS and DB2 for z/OS back out the database updates the program has made since the last commit point. You then restart the program with JCL. If the BMP processes z/OS files, you must provide your own method of taking checkpoints and restarting.

Converting a batch program to a batch-oriented BMP

If you have IMS TM or are running in the DBCTL environment, you can convert a batch program to a batch-oriented BMP.

- If you have IMS TM, you might want to convert your programs for these reasons:
  - BMPs can send output to the message queues.
  - BMPs can access DEDBs and MSDBs.
  - BMPs simplify program recovery because logging goes to a single system log. If you use DASD for the system log in batch, you can specify that you want dynamic backout for the program. In that case, batch recovery is similar to BMP recovery, except, of course, with batch you need to manage multiple logs.
  - Restart can be done automatically from the last checkpoint without changing the JCL.
- If you are using DBCTL, you might want to convert your programs for these reasons:
  - BMPs can access DEDBs.
  - BMPs simplify program recovery because logging goes to a single system log. If you use DASD for the system log in batch, you can specify that you want dynamic backout for the program. In that case, batch recovery is similar to BMP recovery, except, of course, with batch you need to manage multiple logs.
- If you are running sysplex data sharing and you either have IMS TM or are using DBCTL, you might want to convert your program. This is because using batch-oriented BMPs helps you stay within the sysplex data-sharing limit of 32 connections for each OSAM or VSAM structure.

If you use data sharing, you can run batch programs concurrently with online programs. If you do not use data sharing, converting a batch program to a BMP makes it possible to run the program with BMPs and other online programs.

Also, if you plan to run your batch programs offline, converting them to BMPs enables you to run them with the online system, instead of waiting until the online system is not running. Running a batch program as a BMP can also keep the data more current.

- If you have IMS TM or are using DBCTL, you can have a program that runs as either a batch program or a BMP.

  Recommendation: Code your checkpoints in a way that makes them easy to modify. Converting a batch program to a BMP or converting a batch program to use data sharing requires more frequent checkpoints. Also, if a program fails while running in a batch region, you must restart it in a batch region. If a program fails in a BMP region, you must restart it in a BMP region.

The requirements for converting a batch program to a BMP are:

- The program must have an I/O PCB. You can obtain an I/O PCB in batch by specifying the compatibility (CMPAT) option in the program specification block (PSB) for the program.

  Related Reading: For more information on the CMPAT option in the PSB, see *IMS Version 11 System Utilities*.

- BMPs must issue checkpoint calls more frequently than batch programs.

Related concepts:

"When to use checkpoint calls" on page 50

Batch message processing: transaction-oriented BMPs

Transaction-oriented BMPs can access z/OS files, GSAM databases, DB2 for z/OS databases, full-function databases, DEDBs, and MSDBs.

Data a transaction-oriented BMP can access

Unlike a batch-oriented BMP, a transaction-oriented BMP can access the IMS message queue for input and output, and it can only run in the DB/DC and DCCTL environments.

Using a transaction-oriented BMP

Unlike MPPs, transaction-oriented BMPs are not scheduled by IMS. You schedule them as needed and start them with JCL. For example, an MPP, as it processes each message, might send an output message giving details of the transaction to the message queue. A transaction-oriented BMP could then access the message queue to produce a daily activity report.

Typically, you use a transaction-oriented BMP to simulate direct update online: Instead of updating the database while processing its transactions, an MPP sends its updates to the message queue. A transaction-oriented BMP then performs the updates for the MPP. You can run the BMP as needed, depending on the number of updates. This improves response time for the MPP, and it keeps the data current. This can be more efficient than having the MPP process its transactions if the response time of the MPP is very important. One disadvantage in doing this, however, is that it splits the transaction into two parts which is not necessary.

If you have a BMP perform an update for an MPP, design the BMP so that, if the BMP terminates abnormally, you can reenter the last message as input for the BMP when you restart it. For example, suppose an MPP gathers database updates for three BMPs to process, and one of the BMPs terminates abnormally. You would need to reenter the message that the terminating BMP was processing to one of the other BMPs for reprocessing.

BMPs can process transactions defined as wait-for-input (WFI). This means that IMS allows the BMP to remain in virtual storage after it has processed the available input messages. IMS returns a QC status code, indicating that the program should terminate when one of the following occurs:

- The program reaches its time limit.
- The master terminal operator enters a command to stop processing.
- IMS is terminated with a checkpoint shutdown.

You specify WFI for a transaction on the WFI parameter of the TRANSACT macro during IMS system definition.

A batch message processing region (BMP) scheduled against WFI transactions returns a QC status code (no more messages) only for the following commands: `/PSTOP REGION`, `/DBD`, `/DBR`, or `/STA`.

Like MPPs, BMPs can send output messages to several destinations, including other application programs.

Recovering a transaction-oriented BMP

Like MPPs, with transaction-oriented BMPs, you can choose where commit points occur in the program. You can specify that a transaction-oriented BMP be single or multiple mode, just as you can with an MPP. If the BMP is single mode, issuing checkpoint calls is not as critical as in a multiple mode BMP. In a single mode BMP, a commit point occurs each time the program retrieves a message.

Related concepts:

"Identifying output message destinations" on page 106

"When to use checkpoint calls" on page 50

Java message processing: JMPs

A JMP application program is similar to an MPP application program, except that JMP applications must be written in Java or object-oriented COBOL. Like an MPP application, a JMP application is started when there is a message in the message queue for the JMP application and IMS schedules the message for processing.

JMP applications can access IMS data or DB2 for z/OS data using JDBC. JMP applications run in JMP regions which have JVMs (Java Virtual Machines).

Related concepts:

"Overview of the IMS Java dependent regions" on page 641

Java batch processing: JBPs

A JBP application program is similar to a non-message-driven BMP application program, except that JBP applications must be written in Java, object-oriented COBOL, or object-oriented PL/I.

JBP applications can access IMS data or DB2 for z/OS data using JDBC. JBP applications run in JBP regions which have JVMs.

Related concepts:

"Overview of the IMS Java dependent regions" on page 641

IMS programming integrity and recovery considerations

IMS provides support for protecting data integrity for application programs.

How IMS protects data integrity: commit points

When an online program accesses the database, it is not necessarily the only program doing so. IMS and DB2 for z/OS make it possible for more than one application program to access the data concurrently without endangering the integrity of the data.

To access data concurrently while protecting data integrity, IMS and DB2 for z/OS prevent other application programs from accessing segments that your program deletes, replaces, or inserts, until your program reaches a *commit point*. A commit point is the place in the program's processing at which it completes a unit of work. When a unit of work is completed, IMS and DB2 for z/OS commit the changes that your program made to the database. Those changes are now permanent and the changed data is now available to other application programs.

What happens at a commit point

When an application program finishes processing one distinct unit of work, IMS and DB2 for z/OS consider that processing to be valid, even if the program later encounters problems. For example, an application program that is retrieving, processing, and responding to a message from a terminal constitutes a *unit of work*. If the program encounters problems while processing the next input message, the processing it has done on the first input message is not affected. These input messages are separate pieces of processing.

A commit point indicates to IMS that a program has finished a unit of work, and that the processing it has done is accurate. At that time:

- IMS releases segments it has locked for the program since the last commit point. Those segments are then available to other application programs.
- IMS and DB2 for z/OS make the program's changes to the database permanent.
- The current position in all databases except GSAM is reset to the start of the database.

If the program terminates abnormally before reaching the commit point:

- IMS and DB2 for z/OS back out all of the changes the program has made to the database since the last commit point. (This does not apply to batch programs that write their log to tape.)
- IMS discards any output messages that the program has produced since the last commit point.

  Until the program reaches a commit point, IMS holds the program's output messages so that, if the program terminates abnormally, users at terminals and other application programs do not receive inaccurate information from the abnormally terminating application program.

  If the program is processing an input message and terminates abnormally, the input message is **not** discarded if both of the following conditions exist:

1. You are not using the Non-Discardable Messages (NDM) exit routine.
2. IMS terminates the program with one of the following abend codes: U0777, U2478, U2479, U3303. The input message is saved and processed later.

   Exception: The input message is discarded if it is not terminated by one of the abend codes previously referenced. When the program is restarted, IMS gives the program the next message.

If the program is processing an input message when it terminates abnormally, and you use the NDM exit routine, the input message might be discarded from the system regardless of the abend. Whether the input message is discarded from the system depends on how you have written the NDM exit routine.

Related Reading: For more information about the NDM exit routine, see *IMS Version 11 Exit Routines*.

- IMS notifies the MTO that the program terminated abnormally.
- IMS and DB2 for z/OS release any locks that the program has held on data it has updated since the last commit point. This makes the data available to other application programs and users.

Where commit points occur

A commit point can occur in a program for any of the following reasons:

- The program terminates normally. Except for a program that accesses Fast Path resources, normal program termination is always a commit point. A program that accesses Fast Path resources must reach a commit point before terminating.
- The program issues a checkpoint call. Checkpoint calls are a program's means of explicitly indicating to IMS that it has reached a commit point in its processing.
- If a program processes messages as its input, a commit point might occur when the program retrieves a new message. IMS considers this commit point the start of a new unit of work in the program. Retrieving a new message is not always a commit point. This depends on whether the program has been defined as **single mode** or **multiple mode**.
  - If you specify single mode, a commit point occurs each time the program issues a call to retrieve a new message. Specifying single mode can simplify recovery, because you can restart the program from the most recent call for a new message if the program terminates abnormally. When IMS restarts the program, the program begins by processing the next message.
  - If you specify multiple mode, a commit point occurs when the program issues a checkpoint call or when it terminates normally. At those times, IMS sends the program's output messages to their destinations. Because multiple-mode programs contain fewer commit points than do single mode programs, multiple mode programs might offer slightly better performance than single-mode programs. When a multiple mode program terminates abnormally, IMS can only restart it from a checkpoint. Instead of reprocessing only the most recent message, a program might have several messages to reprocess, depending on when the program issued the last checkpoint call.

The following table lists the modes in which the programs can run. Because processing mode is not applicable to batch programs and batch-oriented BMPs, they are not listed in the table. The program type is listed, and the table indicates which mode is supported.

Table 21. Processing modes

Program type	Single mode only	Multiple mode only	Either mode
MPP			X
IFP	X		
Transaction-oriented BMP			X

You specify single or multiple mode on the MODE parameter of the TRANSACT macro.

Related Reading: For information on the TRANSACT macro, see *IMS Version 11 System Definition*.

See the following figure for an illustration of the difference between single-mode and multiple-mode programs. A single-mode program gets and processes messages, sends output, looks for more messages, and terminates if there are no more. A multiple-mode program gets and processes messages, sends output, but has a checkpoint before looking for more messages and terminating. For a single-mode program, the commit points are when the message is obtained and the program terminates. For multiple-mode, the commit point is at the checkpoint and when the program terminates.

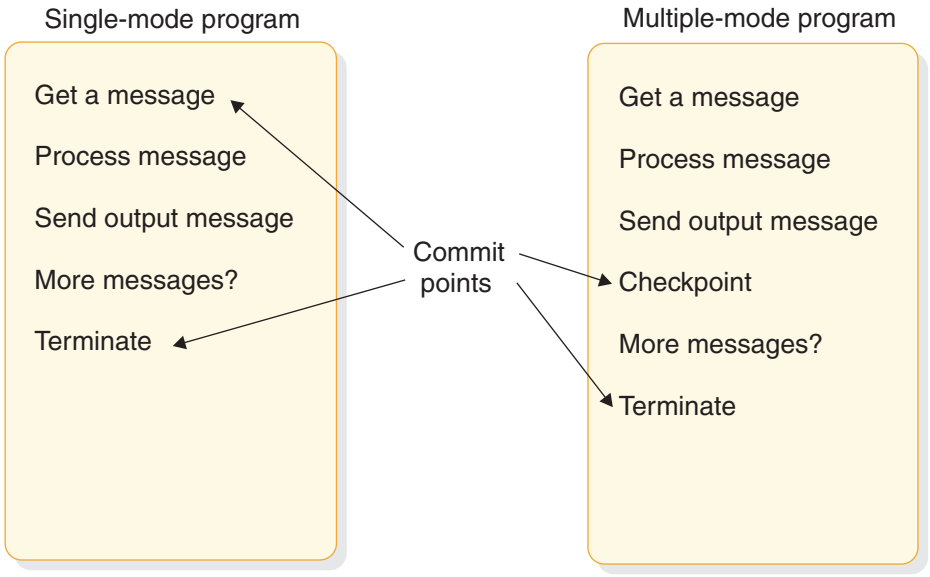


Figure 15. Single mode and multiple mode

DB2 for z/OS does some processing with multiple- and single-mode programs that IMS does not. When a multiple-mode program issues a call to retrieve a new message, DB2 for z/OS performs an authorization check. If the authorization check is successful, DB2 for z/OS closes any SQL cursors that are open. This affects the design of your program.

The DB2 for z/OS SQL `COMMIT` statement causes DB2 for z/OS to make permanent changes to the database. However, this statement is valid only in TSO application programs. If an IMS application program issues this statement, it receives a negative SQL return code.

Planning for program recovery: checkpoint and restart

Recovery in an IMS application program that accesses DB2 for z/OS data is handled by both IMS and DB2 for z/OS. IMS coordinates the process, and DB2 for z/OS handles recovery of DB2 for z/OS data.

Related concepts:

"Introducing checkpoint calls"

"When to use checkpoint calls" on page 50

"Specifying checkpoint frequency" on page 52

Introducing checkpoint calls

Checkpoint calls indicate to IMS that the program has reached a commit point. They also establish places in the program from which the program can be restarted. IMS has symbolic checkpoint calls and basic checkpoint calls.

A program might issue only one type of checkpoint call.

- MPPs and IFPs must use basic checkpoint calls.
- BMP, JMP, and batch programs can use either symbolic checkpoint calls or basic checkpoint calls.

Programs that issue symbolic checkpoint calls can specify as many as seven data areas in the program to be checkpointed. When IMS restarts the program, the Restart call restores these areas to the condition they were in when the program issued the symbolic checkpoint call. Because symbolic checkpoint calls do not support z/OS files, if your program accesses z/OS files, you must supply your own method of establishing checkpoints.

You can use symbolic checkpoint for either Normal Start or Extended Restart (XRST).

For example, typical calls for a Normal start would be as follows:

- XRST (I/O area is blank)
- CHKP (I/O area has checkpoint ID)
- Database Calls (including checkpoints)
- CHKP (final checkpoint)

For example, typical calls for an Extended Restart (XRST) would be as follows:

- XRST (I/O area has checkpoint ID)
- CHKP (I/O area has new checkpoint ID)
- Database Calls (including checkpoints)
- CHKP (final checkpoint)

The restart call, which you must use with symbolic checkpoint calls, provides a way of restarting a program after an abnormal termination. It restores the program's data areas to the way they were when the program issued the symbolic checkpoint call. It also restarts the program from the last checkpoint the program established before terminating abnormally.

All programs can use basic checkpoint calls. Because you cannot use the restart call with the basic checkpoint call, you must provide program restart. Basic checkpoint calls do not support either z/OS or GSAM files. IMS programs cannot use z/OS checkpoint and restart. If you access z/OS files, you must supply your own method of establishing checkpoints and restarting.

In addition to the actions that occur at a commit point, issuing a checkpoint call causes IMS to:

- Inform DB2 for z/OS that the changes your program has made to the database can be made permanent. DB2 for z/OS makes the changes to DB2 for z/OS data permanent, and IMS makes the changes to IMS data permanent.
- Write a log record containing the checkpoint identification given in the call to the system log, but only if the PSB contains a DB PCB. You can print checkpoint log records by using the IMS File Select and Formatting Print program (DFSERA10). With this utility, you can select and print log records based on their type, the data they contain, or their sequential positions in the data set. Checkpoint records are X'18' log records.

  Related Reading: For more information about the DFSERA10 program, see *IMS Version 11 System Utilities*.
- Send a message containing the checkpoint identification that was given in the call to the system console operator and to the IMS master terminal operator.
- Return the next input message to the program's I/O area, if the program processes input messages. In MPPs and transaction-oriented BMPs, a checkpoint call acts like a call for a new message.

Restriction: Do not specify CHKPT=EOV on any DD statement in order to take an IMS checkpoint because of unpredictable results.

Related concepts:

"Planning for program recovery: checkpoint and restart" on page 49

When to use checkpoint calls

Issuing Checkpoint calls is most important in programs that do not have built-in commit points.

The decision about whether your program should issue checkpoints, and if so, how often, depends on your program. Generally, these programs should issue checkpoint calls:

- Multiple-mode programs
- Batch-oriented BMPs (which can issue either `SYNC` or `CHKP` calls)
- Most batch programs
- Programs that run in a data sharing environment
- JMP applications

You do not need to issue checkpoint calls in:

- Single-mode BMP or MPP programs
- Database load programs
- Programs that access the database in read-only mode, as defined with the PROCOPT=GO option (during a PSBGEN), and are short enough to restart from the beginning
- Programs that have exclusive use of the database

Checkpoints in MPPs and transaction-oriented BMPs

The mode type of the program is specified on the MODE keyword of the TRANSACT macro during IMS system generation. The modes are single and multiple.

- **In single-mode programs**

In single mode programs (MODE=SNGL was specified on the TRANSACT macro during IMS system definition), a Get Unique to the message queue causes an implicit commit to be performed.

- **In multiple-mode programs**

  In multiple-mode BMPs and MPPs, the only commit points are those that result from the checkpoint calls that the program issues and from normal program termination. If the program terminates abnormally and it has not issued checkpoint calls, IMS backs out the program's database updates and cancels the messages it created since the beginning of the program. If the program has issued checkpoint calls, IMS backs out the program's changes and cancels the output messages it has created since the most recent checkpoint.

  Consider the following when issuing checkpoint calls in multiple-mode programs:

  – How long it would take to back out and recover that unit of processing. The program should issue checkpoints frequently enough to make the program easy to back out and recover.

  – How you want the output messages grouped. checkpoint calls establish how a multiple-mode program's output messages are grouped. Programs should issue checkpoint calls frequently enough to avoid building up too many output messages.

  Depending on the database organization, issuing a checkpoint call might reset your position in the database.

  Related Reading: For more information about losing your position when a checkpoint is issued, see *IMS Version 11 Database Administration*.

Checkpoints in batch-oriented BMPs

Issuing checkpoint calls in a batch-oriented BMP is important for several reasons:

- In addition to committing changes to the database and establishing places from which the program can be restarted, checkpoint calls release resources that IMS has locked for the program.
- A batch-oriented BMP that uses DEDBs or MSDBs might terminate with abend U1008 if a `SYNC` or `CHKP` call is not issued before the application program terminates.
- If a batch-oriented BMP does not issue checkpoints frequently enough, it can be abnormally terminated, or it can cause another application program to be abnormally terminated by IMS for any of these reasons:

  – If a BMP retrieves and updates many database records between checkpoint calls, it can tie up large portions of the databases and cause long waits for other programs needing those segments.

    Exception: For a BMP with a processing option of GO or exclusive, IMS does not lock segments for programs. Issuing checkpoint calls releases the segments that the BMP has locked and makes them available to other programs.

  – The space needed to maintain lock information about the segments that the program has read and updated exceeds what has been defined for the IMS system. If a BMP locks too many segments, the amount of storage needed for the locked segments can exceed the amount of available storage. If this happens, IMS terminates the program abnormally. You must increase the program's checkpoint frequency before rerunning the program. The available storage is specified during IMS system definition.

Related Reading: For more information on specifying storage, see *IMS Version 11 System Definition*.

You can limit the number of locks for the BMP by using the LOCKMAX=*n* parameter on the PSBGEN statement. For example, a specification of LOCKMAX=5 means the application cannot obtain more than 5000 locks at any time. The value of *n* must be between 0 and 255. When a maximum lock limit does not exist, 0 is the default. If the BMP tries to acquire more than the specified number of locks, IMS terminates the application with abend U3301.

Related Reading: For more information about this abend, see *IMS Version 11 Messages and Codes, Volume 3: IMS Abend Codes*.

Checkpoints in batch programs

Batch programs that update databases should issue checkpoint calls. The main consideration in deciding how often to take checkpoints in a batch program is the time required to back out and reprocess the program after a failure. A general recommendation is to issue one checkpoint call every 10 or 15 minutes.

If you might need to back out the entire batch program, the program should issue the checkpoint call at the beginning of the program. IMS backs out the program to the checkpoint you specify, or to the most recent checkpoint, if you do not specify a checkpoint. If the database is updated after the beginning of the program and before the first checkpoint, IMS is not able to back out these database updates.

For a batch program to issue checkpoint calls, it must specify the compatibility option in its PSB (CMPAT=YES). This generates an I/O PCB for the program, which IMS uses as an I/O PCB in the checkpoint call.

Another important reason for issuing checkpoint calls in batch programs is that, although they may currently run in an IMS batch region, they might later need to access online databases. This would require converting them to BMPs. Issuing checkpoint calls in a BMP is important for reasons other than recovery—for example, to release database resources for other programs. So, you should initially include checkpoints in all batch programs that you write. Although the checkpoint support might not be needed then, it is easier to incorporate checkpoint calls initially than to try to fit them in later.

To free database resources for other programs, batch programs that run in a data-sharing environment should issue checkpoint calls more frequently than those that do not run in a data-sharing environment.

Related concepts:

"DB batch processing" on page 39

"Batch processing online: batch-oriented BMPs" on page 42

"Batch message processing: transaction-oriented BMPs" on page 44

"Planning for program recovery: checkpoint and restart" on page 49

Specifying checkpoint frequency

You should specify checkpoint frequency in your program so that you can easily modify it when the frequency needs to be adjusted.

You can do this by:

- Using a counter in your program to keep track of elapsed time, and issuing a checkpoint call after a certain time interval.

- Using a counter to keep track of the number of root segments your program accesses, and issuing a checkpoint call after a certain number of root segments.
- Using a counter to keep track of the number of updates your program performs, and issuing a checkpoint call after a certain number of updates.

Related concepts:

"Planning for program recovery: checkpoint and restart" on page 49

Data availability considerations

The following information describes the conditions that could cause data to become unavailable in a full-function database and the program calls that allow your program to manage data under these conditions.

Dealing with unavailable data

The conditions that make the database unavailable for both read and update are:

- The `/LOCK` command for a database was issued.
- The `/STOP` command for a database was issued.
- The `/DBRECOVERY` command was issued.
- Authorization for a database failed.

The conditions that make the database available only for read and not for update are:

- The `/DBDUMP` command has been issued.
- Database ACCESS value is RD (read).

In addition to unavailability of an entire database, other situations involving unavailability of a limited amount of data can also inhibit program access. One such example would be a failure situation involving data sharing. The active IMS system knows which locks were held by a sharing IMS system at the time the sharing IMS system failed. Although the active IMS system continues to use the database, it must reject access to the data which the failed IMS system locked upon failure. This situation occurs for both full-function and DEDB databases.

The two situations where the program might encounter unavailable data are:

- The program makes a call requiring access to a database that was unavailable at the time the program was scheduled.
- The database was available when the program was scheduled, but limited amounts of data are unavailable. The current call has attempted to access the unavailable data.

Regardless of the condition causing the data to be unavailable, the program has two possible approaches when dealing with unavailable data. The program can be **insensitive** or **sensitive** to data unavailability.

- When the program is insensitive, IMS takes appropriate action when the program attempts to access unavailable data.
- When the program is sensitive, IMS informs the program that the data it is attempting to access is not available.

If the program is insensitive to data unavailability, and attempts to access unavailable data, IMS aborts the program (3303 pseudo-abend), and backs out any updates the program has made. The input message that the program was

processing is suspended, and the program is scheduled to process the input message when the data becomes available. However, if the database is unavailable because dynamic allocation failed, a call results in an AI (unable to open) status code.

If the program is sensitive to data unavailability and attempts to access unavailable data, IMS returns a status code indicating that it could not process the call. The program then takes the appropriate action. A facility exists for the program to initiate the same action that IMS would have taken if the program had been insensitive to unavailable data.

IMS does not schedule batch programs if the data that the program can access is unavailable. If the batch program is using block-level data sharing, it might encounter unavailable data if the sharing system fails and the batch system attempts to access data that was updated but not committed by the failed system.

The following conditions alone do not cause a batch program to fail during initialization:

- A PCB refers to a HALDB.
- The use of DBRC is suppressed.

However, without DBRC, a database call using a PCB for a HALDB is not allowed. If the program is sensitive to unavailable data, such a call results in the status code BA; otherwise, such a call results in message DFS3303I, followed by ABENDU3303.

Scheduling and accessing unavailable databases

By using the INIT, INQY, SETS, SETU, and ROLS calls, the program can manage a data environment where the program is scheduled with unavailable databases.

The INIT call informs IMS that the program is sensitive to unavailable data and can accept the status codes that are issued when the program attempts to access such data. The INIT call can also be used to determine the data availability for each PCB.

The INQY call is operable in both batch and online IMS environments. IMS application programs can use the INQY call to request information regarding output destination, session status, the current execution environment, the availability of databases, and the PCB address based on the PCBNAME. The INQY call is only supported by way of the AIB interface (AIBTDLI or CEETDLI using the AIB rather than the PCB address).

The SETS, SETU, and ROLS calls enable the application to define multiple points at which to preserve the state of full-function (except HSAM) databases and message activity. The application can then return to these points at a later time. By issuing a SETS or SETU call before initiating a set of DL/I calls to perform a function, the program can later issue the ROLS call if it cannot complete a function due to data unavailability.

The ROLS call allows the program to roll back its IMS full-function database activity to the state that it was in prior to a SETS or SETU call being issued. If the PSB contains an MSDB or a DEDB, the SETS and ROLS (with token) calls are invalid. Use the SETU call instead of the SETS call if the PSB contains a DEDB, MSDB, or GSAM PCB.

The ROLS call can also be used to undo all update activity (database and messages) since the last commit point and to place the current input message on the suspend queue for later processing. This action is initiated by issuing the ROLS call without a token or I/O area.

Restriction: With DB2 for z/OS, you cannot use ROLS (with a token) or SETS.

Related information:

3303 (Messages and Codes)

Use of STAE or ESTAE and SPIE in IMS programs

IMS uses STAE or ESTAE routines in the control region, the dependent (MPP, IFP, BMP) regions, and the batch regions. In the control region, STAE or ESTAE routines ensure that database logging and various resource cleanup functions are complete.

In the dependent region, STAE or ESTAE routines are used to notify the control region of any abnormal termination of the application program or the dependent region itself. If the control region is not notified of the dependent region termination, resources are not properly released and normal checkpoint shutdown might be prevented.

In the batch region, STAE or ESTAE routines ensure that database logging and various resource cleanup functions are complete. If the batch region is not notified of the application program termination, resources might not be properly released.

Two important aspects of the STAE or ESTAE facility are that:

- IMS relies on its STAE or ESTAE facility to ensure database integrity and resource control.
- The STAE or ESTAE facility is also available to the application program.

Because of these two factors, be sure you clearly understand the relationship between the program and the STAE or ESTAE facility.

Generally, do not use the STAE or ESTAE facility in your application program. However, if you believe that the STAE or ESTAE facility is required, you must observe the following basic rules:

- When the environment supports STAE or ESTAE processing, the application program STAE or ESTAE routines always get control before the IMS STAE or ESTAE routines. Therefore, you must ensure that the IMS STAE or ESTAE exit routines receive control by observing the following procedures in your application program:
  - Establish the STAE or ESTAE routine only once and always before the first DL/I call.
  - When using the STAE or ESTAE facility, the application program should not alter the IMS abend code.
  - Do not use the RETRY option when exiting from the STAE or ESTAE routine. Instead, return a CONTINUE-WITH-TERMINATION indicator at the end of the STAE or ESTAE processing. If your application program specifies the RETRY option, be aware that IMS STAE or ESTAE exit routines will not get control to perform cleanup. Therefore, system and database integrity might be compromised.
- The application program STAE or ESTAE exit routine must not issue DL/I calls (DB or TM) because the original abend might have been caused by a problem

between the application and IMS. A problem between the application and IMS could result in recursive entry to STAE or ESTAE with potential loss of database integrity, or in problems taking a checkpoint. This also could result in a hang condition or an ABENDU0069 during termination.

Related concepts:

"What to do when your IMS program terminates abnormally" on page 165

Dynamic allocation for IMS databases

Use the dynamic allocation function to specify the JCL information for IMS databases in a library instead of in the JCL of each batch or online job.

If you use dynamic allocation, do not include JCL DD statements for any database data sets that have been defined for dynamic allocation. Check with the DBA or comparable specialist to determine which databases have been defined for dynamic allocation.

Related Reading: For additional information on the definitions for dynamic allocation, see the description of the DFSMDA macro in *IMS Version 11 System Definition*.

Chapter 4. Analyzing CICS application processing requirements

IMS supports application programs running in a CICS environment

Defining CICS application requirements

One of the steps of application design is to decide how the business processes, or tasks can be best grouped into a set of programs that will efficiently perform the required processing.

Some of the considerations in analyzing processing requirements are:

- **When the task must be performed**
  - Will it be scheduled unpredictably (for example on terminal demand) or periodically (for example, weekly)?
- **How the program that performs the task is executed**
  - Will it be executed online, where response time is more important, or by batch job submission, where a slower response time is acceptable?
- **The consistency of the processing components**
  - Does this action the program is to perform involve more than one type of program logic? For example, does it involve mostly retrievals, and only one or two updates? If so, you should consider separating the updates into a separate program.
  - Does this action involve several large groups of data? If it does, it might be more efficient to separate the programs by the data they access.
- **Any special requirements about the data or processing**

  Security
  Should access to the program be restricted?

  Recovery
  Are there special recovery considerations in the program's processing?

  Integrity
  Do other departments use the same data?

Answers to questions like these can help you decide on the number of application programs that the processing will require, and on the types of programs that perform the processing most efficiently. Although rules dealing with how many programs can most efficiently do the required processing do not exist, here are some suggestions:

- As you look at each programming task, examine the data and processing that each task involves. If a task requires different types of processing and has different time limitations (for example, weekly as opposed to monthly), that task may be more efficiently performed by several programs.
- As you define each program, it is a good idea for maintenance and recovery reasons to keep programs as simple as possible. The simpler a program is—the less it does—the easier it is to maintain, and to restart after a program or system failure. The same is true with data availability—the less data that is accessed, the more likely the data is to be available; the more limited the data accessed, the more likely the data is to be available.

© Copyright IBM Corp. 1974, 2015

Similarly, if the data that the application requires is physically in one place, it might be more efficient to have one program do more of the processing than usual. These are considerations that depend on the processing and the data of each application.

- Documenting each of the user tasks is helpful during the design process, and in the future when others will work with your application. Be sure you are aware of the standards in this area. The kind of information that is typically kept is when the task is to be executed, a functional description, and requirements for maintenance, security, and recovery.

  For example, for the Current Roster process described previously, you might record the information shown in the following form. How frequently the program is run is determined by the number of classes (20) for which the Ed Center will print current rosters each week.

Example: Current roster task description

```
                    USER TASK DESCRIPTION

NAME: Current Roster
ENVIRONMENT: Batch                    FREQUENCY: 20 per week

INVOKING EVENT OR DOCUMENT: Time period (one week)

REQUIRED RESPONSE TIME: 24 hours

FUNCTION DESCRIPTION: Print weekly, a current student roster, in student
 number sequence for each class offered at the Education Center.

MAINTENANCE: Included in Education DB maintenance.

SECURITY: None.

RECOVERY: After a failure, the ability to start printing a particular
class roster starting from a particular sequential student number.
```

Accessing databases with your CICS application program

When designing your program, consider the type of data it must access. The type of data depends on the operating environment.

The data from IMS and DB2 for z/OS databases, and z/OS files, that is available to CICS online and IMS batch programs is shown in the following table.

Table 22. The data that your CICS program can access

Type of program	IMS databases	DB2 for z/OS databases	z/OS files
CICS online	Yes[1]	Yes[2]	Yes[3]
DB batch	Yes	Yes[3]	Yes

Notes:

1. Except for Generalized Sequential Access Method (GSAM) databases. GSAM enables batch programs to access a sequential z/OS data set as a simple database.
2. IMS does not participate in the call process.
3. Access through CICS file control or transient data services.

Also, consider the type of database your program must access. As shown in the following table, the type of program you can write and database that can be accessed depends on the operating environment.

Table 23. Program and database options in the CICS environments

Environment[1]	Type of program you can write	Type of database that can be accessed
DB batch	DB batch	DB2 for z/OS[2] DL/I Full-function GSAM z/OS files
DBCTL	BMP	DB2 for z/OS DEDBs Full-function GSAM z/OS files
	CICS online	DB2 for z/OS[2] DEDBs Full-function z/OS files (access through CICS file control or transient data services)

Notes:

1. A CICS environment, or CICS remote DL/I environment also exists and is also referred to as function shipping. In this environment, a CICS system supports applications that issue DL/I calls but the CICS system does not service the requests itself. The CICS environment "function ships" the DL/I calls to another CICS system that is using DBCTL. For more information on remote DL/I, see *CICS Transaction Server for z/OS IMS Database Control Guide*.
2. IMS does not participate in the call process.

The types of databases that can be accessed are:

- **Full-Function Databases**

  Full-function databases are hierarchic databases that are accessed through Data Language I (DL/I). DL/I calls enable application programs to retrieve, replace, delete, and add segments to full-function databases. CICS online and BMP programs can access the same database concurrently (if participating in IMS data sharing); an IMS batch program must have exclusive access to the database (if not participating in IMS data sharing).

  All types of programs (batch, BMPs, and online) can access full-function databases.

- **Fast Path DEDBs**

  Data entry databases (DEDBs) are hierarchic databases for, and efficient access to, large volumes of detailed data. In the DBCTL environment, CICS online and BMP programs can access DEDBs.

- **DB2 for z/OS Databases**

  DB2 for z/OS databases are relational databases. Relational databases are represented to application programs and users as tables and are processed using a relational data language called Structured Query Language (SQL). DB2 for z/OS databases can be processed by CICS online transactions, and by IMS batch and BMP programs.

  Related Reading: For information on processing DB2 for z/OS databases, see *DB2 for z/OS Application Programming and SQL Guide*.

- **GSAM Databases**

Generalized Sequential Access Method (GSAM) is an access method that enables BMPs and batch programs to access a "flat" sequential z/OS data set as a simple database. A GSAM database can be accessed by z/OS or CICS.

- **z/OS Files**

  CICS online and IMS batch programs can access z/OS files for their input, processing, or output. Batch programs can access z/OS files directly; online programs must access them through CICS file control or transient data services.

Related concepts:

"Using data sharing for your CICS program" on page 61

Writing a CICS program to access IMS databases

The types of programs you can use depend on whether you are running in the DBCTL environment. Within the different environments, the type of program you write depends on the processing your application requires. Each type of program answers different application requirements.

Related concepts:

Chapter 35, "IMS solutions for Java development overview," on page 539

Writing a CICS online program

Use the following information to decide if an online program is appropriate for your application.

Data that a CICS online program can access

CICS online programs run in the DBCTL environment and can access IMS full-function databases, Fast Path DEDBs, DB2 for z/OS databases, and z/OS files.

Online programs that access IMS databases are executed in the same way as other CICS programs.

Using a CICS online program

An online program runs under the control of CICS, and it accesses resources concurrently with other online programs. Some of the application requirements online programs can answer are:

- Information in the database must be available to many users.
- Program needs to communicate with terminals and other programs.
- Programs must be available to users at remote terminals.
- Response time is important.

The structure of an online program, and the way it receives status information, depend on whether it is a call- or command-level program. However, both command- and call-level online programs:

- Schedule a PSB (for CICS online programs). A PSB is automatically scheduled for batch or BMP programs.
- Issue either commands or calls to access the database. Online programs cannot mix commands and calls in one logical unit of work (LUW).
- Optionally, terminate a PSB for CICS online programs.

- Issue an `EXEC CICS RETURN` statement when they have finished their processing. This statement returns control to the linking program. When the highest-level program issues the RETURN statement, CICS regains control and terminates the PSB if it has not yet been terminated.

Because an online application program can be used concurrently by several tasks, it must be quasi-reentrant.

An online program in the DBCTL environment can use many IMS system service requests.

DL/I database or system service requests must refer to one of the program communication blocks (PCBs) from the list of PCBs passed to your program by IMS. The PCB that must be used for making system service requests is called the I/O PCB. When present, it is the first PCB in the list of PCBs.

For an online program in the DBCTL environment, the I/O PCB is optional. To use the I/O PCB, you must indicate this in the application program when it schedules the PSB.

Before you run your program, the program specification blocks (PSBs) and database descriptions (DBDs) the program uses must be converted to internal control block format using the IMS ACBGEN utility. PSBs specify the characteristics of an application program. DBDs specify the physical and logical characteristics of IMS databases.

Related Reading: For more information on performing an ACBGEN and a PSBGEN, see *IMS Version 11 System Utilities*.

Because an online program shares a database with other online programs, it may affect the performance of your online system.

Related concepts:

"Maximizing the performance of your CICS system" on page 63

"Distributed and local connectivity with the IMS Universal drivers" on page 550

Related tasks:

"Programming Java applications for CICS with the classic Java APIs for IMS" on page 676

Using data sharing for your CICS program

If you use data sharing, your programs can participate in IMS data sharing. Under data sharing, CICS online and BMP programs can access the same DL/I database concurrently.

Batch programs in a data-sharing environment can access databases used by other batch programs, and by CICS and IMS online programs. With data sharing, you can share data directly and your program's requests need not go through a mirror transaction.

Related Reading: For more information on sharing a database with an IMS system, see *IMS Version 11 System Administration*.

Related concepts:

"Accessing databases with your CICS application program" on page 58

Scheduling and terminating a PSB (CICS online programs only)

Before your online program issues any DL/I calls, it must indicate to IMS its intent to use a particular PSB by issuing either a `PCB` call or a `SCHD` command. In addition to indicating which PSB your program will use, the `PCB` call obtains the address of the PCBs in the PSB. When you no longer need a PSB, you can terminate it using the `TERM` request.

In a CICS online program, you use a `PCB` call or `SCHD` command (for command-level programs) to obtain the PSB for your program. Because CICS releases the PSB your program uses when the transaction ends, your program need not explicitly terminate the PSB. Only use a terminate request if you want to:

- Use a different PSB
- Commit all the database updates and establish a logical unit of work for backing out updates
- Free IMS resources for use by other CICS tasks

A terminate request causes a CICS sync point, and a CICS sync point terminates the PSB. For more information about CICS recovery concepts, see the appropriate CICS publication.

Do not use terminate requests for other reasons because:

- A terminate request forces a CICS sync point. This sync point releases all recoverable resources and IMS database resources that were enqueued for this task.

  If the program continues to update other CICS resources after the terminate request and then terminates abnormally, only those resources that were updated after the terminate request are backed out. Any IMS changes made by the program are not backed out.
- IMS lock management detects deadlocks that occur if two transactions are waiting for segments held by the other.

  When a deadlock is detected, one transaction is abnormally terminated. Database changes are backed out to the last `TERM` request. If a `TERM` request or CICS sync point was issued prior to the deadlock, CICS does not restart the transaction.

  Related Reading: For a complete description of transaction restart considerations, see *CICS Transaction Server for z/OS Recovery and Restart Guide*.
- Issuing a terminate request causes additional logging.
- If the terminal output requests are issued after a terminate request and the transaction fails at this point, the terminal operator does not receive the message.

  The terminal operator may assume that the entire transaction failed, and reenter the input, thus repeating the updates that were made before the terminate request. These updates were not backed out.

Linking and passing control to other programs (CICS online programs only)

Use CICS to link your program to other programs without losing access to the facilities acquired in the linking program.

For example:

- You could schedule a PSB and then link to another program using a `LINK` command. On return from that program, the PSB is still scheduled.
- Similarly, you could pass control to another program using the `XCTL` command, and the PSB remains scheduled until that program issues an EXEC CICS RETURN statement. However, when you pass control to another program using `XCTL`, the working storage of the program passing control is lost. If you want to retain the working storage for use by the program being linked to, you must pass the information in the COMMAREA.

Recommendation: To simplify your work, instead of linking to another program, you can issue all DL/I requests from one program module. This helps to keep the programming simple and easy to maintain.

Terminating a PSB or issuing a sync point affects the linking program. For example, a terminate request or sync point that is issued in the program that was linked causes the release of CICS resources enqueued in the linking program.

How CICS distributed transactions access IMS

CICS can divide a single, logical unit of work into separate CICS transactions and coordinate the sync point globally. If such CICS transactions access DBCTL, locking and buffer management issues might occur.

To IMS, the transactions are separate units of work, on different DBCTL threads, and they do not share locks or buffers. For example, if a global transaction runs, obtains a database lock, and reaches the commit point, CICS does not process the synchronization point until the other transactions in the CICS unit of recovery (UOR) are ready to commit. If a second transaction in the same CICS UOR requests the same lock as that held by the first transaction, the second transaction is held in a lock wait state. The first transaction cannot complete the sync point and release the lock until the second transaction also reaches the commit point, but this cannot happen because the second transaction is in a lock wait state. You must ensure that this type of collision does not occur with CICS distributed transactions that access IMS.

Maximizing the performance of your CICS system

When you write programs that share data with other programs (for example, a program that will participate in IMS data sharing or a BMP), be aware of how your program affects the performance of the online system.

A BMP program, in particular, can affect the performance of the CICS online transactions. This is because BMP programs usually make a larger number of database updates than CICS online transactions, and a BMP program is more likely to hold segments that CICS online programs need. Limit the number of segments held by a BMP program, so CICS online programs need not wait to acquire them.

One way to limit the number of segments held by a BMP or batch program that participates in IMS data sharing is to issue checkpoint requests in your program to commit database changes and release segments held by the program. When deciding how often to issue checkpoint requests, you can use one or more of the following techniques:

- Divide the program into small logical units of work, and issue a checkpoint call at the end of each unit.

- Issue a checkpoint call after a certain number of DL/I requests have been issued, or after a certain number of transactions are processed.

In CICS online programs, release segments for use by other transactions to maximize the performance of your online system. (Ordinarily, database changes are committed and segments are released only when control is returned to CICS.) To more quickly free resources for use by other transactions, you can issue a `TERM` request to terminate the PSB. However, less processing overhead generally occurs if the PSB is terminated when control is returned to CICS.

Related concepts:

"Writing a CICS online program" on page 60

"Taking checkpoints in batch and BMP programs" on page 65

Related tasks:

"Programming Java applications for CICS with the classic Java APIs for IMS" on page 676

Programming integrity and database recovery considerations for your CICS program

IMS provides support for protecting data integrity for CICS online programs

How IMS protects data integrity for CICS online programs

IMS can protect the data integrity for CICS online programs.

IMS protects the integrity of the database for programs that share data by:

- Preventing other application programs with update capability from accessing any segments in the database record your program is processing, until your program finishes with that record and moves to a new database record in the same database.
- Preventing other application programs from accessing segments that your program deletes, replaces, or inserts, until your program reaches a sync point. When your program reaches a sync point, the changes your program has made to the database become permanent, and the changed data becomes available to other application programs.

  Exception: If PROCOPT=GO has been defined during PSBGEN for your program, your program can access segments that have been updated but not committed by another program.
- Backing out database updates made by an application program that terminates abnormally.

You may also want to protect the data your program accesses by retaining segments for the sole use of your program until your program reaches a sync point—even if you do not update the segments. (Ordinarily, if you do not update the segments, IMS releases them when your program moves to a new database record.) You can use the Q command code to reserve segments for the exclusive use of your program. You should use this option only when necessary because it makes data unavailable to other programs and can have an impact on performance.

Recovering databases accessed by batch and BMP programs

You can plan for recovering databases accessed by batch or BMP programs.

CICS recovers databases accessed by CICS online programs in the same way it handles other recoverable CICS resources. For example, if an IMS transaction terminates abnormally, CICS and IMS back out all database updates to the last sync point.

For batch or BMP programs, do the following:

- Take checkpoints in your program to commit database changes and provide places from which your program can be restarted.
- Provide the code for or issue a request to restart your program.

You may also want to back out the database changes that have been made by a batch program that has not yet committed these changes.

To perform these tasks, you use system service calls, described in more detail in the appropriate application programming information for your environment.

Requesting an I/O PCB in batch programs

For your program to successfully issue any system service request, an I/O PCB must have been previously requested.

Related concepts:

"Developing JBP applications with the IMS Java dependent region resource adapter" on page 652

Taking checkpoints in batch and BMP programs

You can take checkpoints in batch and BMP programs. Checkpoints are important for recovery and for integrity.

Taking checkpoints in batch and BMP programs is important for two reasons:

- Recovery: Checkpoints establish places in your program from which your program could be restarted, in the event of a program or system failure. If your program abnormally terminates after issuing a checkpoint request, database changes will be backed out to the point at which the checkpoint request was issued.
- Integrity: Checkpoints also commit the changes that your program has made to the database.

In addition to providing places from which to restart your program and committing database changes, issuing checkpoint calls in a BMP program or in a program participating in IMS data sharing releases database segments for use by other programs.

When a batch or BMP program issues a checkpoint request, IMS writes a record containing a checkpoint ID to the IMS system log.

When your application program reaches a point during its execution where you want to make sure that all changes made to that point have been physically entered in the database, issue a checkpoint request. If some condition causes your program to fail before its execution is complete, the database must be restored to its original state. The changes made to the database must be backed out so that the database is not left in a partially updated condition for access by other application programs.

If your program runs a long time, you can reduce the number of changes that must be backed out by taking checkpoints in your program. Then, if your program

terminates abnormally, only the database updates that occurred after the checkpoint must be backed out. You can also restart the program from the point at which you issued the checkpoint request, instead of having to restart it from the beginning.

Issuing a checkpoint call cancels your position in the database.

Issue a checkpoint call just before issuing a Get Unique call, which reestablishes your position in the database record after the checkpoint is taken.

Types of checkpoints

The two types of checkpoint calls are basic and symbolic. Both types commit your program's changes to the database and establish places from which your program can be restarted:

Batch and BMP programs can issue basic checkpoint calls using the `CHKP` call. When you use basic checkpoint calls, you must provide the code for restarting the program after an abnormal termination.

Batch and BMP programs can also issue symbolic checkpoint calls. You can issue a symbolic checkpoint call by using the `CHKP` call. Like the basic checkpoint call, the symbolic checkpoint call commits changes to the database and establishes places from which the program can be restarted. In addition, the symbolic checkpoint call:

- Works with the Extended Restart call to simplify program restart and recovery.
- Lets you specify as many as seven data areas in the program to be checkpointed. When you restart the program, the restart call restores these areas to the way they were when the program terminated abnormally.

Specifying a checkpoint ID

Each checkpoint call your program issues must have an identification, or ID. Checkpoint IDs must be 8 bytes in length and contain printable EBCDIC characters.

When you want to restart your program, you can supply the ID of the checkpoint from which you want the program to be started. This ID is important because when your program is restarted, IMS searches for checkpoint information with an ID matching the one you have supplied. The first matching ID that IMS finds becomes the restart point for your program. This means that checkpoint IDs must be unique both within each application program and among application programs. If checkpoint IDs are not unique, you cannot be sure that IMS will restart your program from the checkpoint you specified.

One way to make sure that checkpoint IDs are unique within and among programs is to construct IDs in the following order:

- Three bytes of information that uniquely identifies your program.
- Five bytes of information that serves as the ID within the program, for example, a value that is increased by 1 for each checkpoint command or call, or a portion of the system time obtained at program start by issuing the TIME macro.

Specifying checkpoint frequency

To determine the frequency of checkpoint requests, you must consider the type of program and its performance characteristics.

In batch programs

When deciding how often to issue checkpoint requests in a batch program, you should consider the time required to back out and reprocess the program after a failure. For example, if you anticipate that the processing your program performs will take a long time to back out, you should establish checkpoints more frequently.

If you might back out of the entire program, issue the checkpoint request at the very beginning of the program. IMS backs out the database updates to the checkpoint you specify. If the database is updated after the beginning of the program and before the first checkpoint, IMS is not able to back out these database updates.

In a data-sharing environment, also consider the impact of sharing resources with other programs on your online system. You should issue checkpoint calls more frequently in a batch program that shares data with online programs, to minimize resource contention.

It is a good idea to design all batch programs with checkpoint and restart in mind. Although the checkpoint support may not be needed initially, it is easier to incorporate checkpoint calls initially than to try to fit them in later. If the checkpoint calls are incorporated, it is easier to convert batch programs to BMP programs or to batch programs that use data sharing.

In BMP programs

When deciding how often to issue checkpoint requests in a BMP program, consider the performance of your CICS online system. Because these programs share resources with CICS online transactions, issue checkpoint requests to release segments so CICS online programs need not wait to acquire them.

Printing checkpoint log records

You can print checkpoint log records by using the IMS File Select and Formatting Print Program (DFSERA10). With this utility, you can select and print log records based on their type, the data they contain, or their sequential positions in the data set. Checkpoint records are type 18 log records. *IMS Version 11 System Utilities* describes this program.

Related concepts:

"Maximizing the performance of your CICS system" on page 63

Backing out database changes

If your program terminates abnormally, the database must be restored to its previous state and uncommitted changes must be backed out. Changes made by a BMP or CICS online program are automatically backed out. Database changes made by a batch program might or might not be backed out, depending on whether your system log is on DASD.

For a batch program

What happens when a batch program terminates abnormally and how you recover the database depend on the storage medium for the system log. You can specify that the system log is to be stored on either DASD or on tape.

- **When the system log is on DASD**

You can specify that IMS is to dynamically back out the changes that a batch program has made to the database since its last commit point by coding BKO=Y in the JCL. IMS performs dynamic backout for a batch program when an IMS-detected failure occurs, such as when a deadlock is detected (for batch programs that share data).

DASD logging also makes it possible for batch programs to issue the rollback (ROLB) system service request, in addition to ROLL. The ROLB request causes IMS to dynamically back out the changes the program has made to the database since its last commit point, and then to return control to the application program.

Dynamically backing out database changes has the following advantages:

- Data accessed by the program that failed is immediately available to other programs. Otherwise, if batch backout is not used, data is not available to other programs until the IMS Batch Backout utility has been run to back out the database changes.
- If two programs are deadlocked, one of the programs can continue processing. Otherwise, if batch backout is not used, both programs will fail. (This applies only to batch programs that share data.)

Instead of using dynamic backout, you can run the IMS Batch Backout utility to back out changes.

- **When the system log is on tape**

  If a batch application program terminates abnormally and the system log is stored on tape, you must use the IMS Batch Backout utility to back out the program's changes to the database.

Related Reading: For more information, see *IMS Version 11 Database Utilities*.

For BMP programs

If your program terminates abnormally, the changes the program has made since the last commit point are backed out. If a system failure occurs, or if the CICS control region or DBCTL terminates abnormally, DBCTL emergency restart backs out all changes made by the program since the last commit point. You need not use the IMS Batch Backout utility because DBCTL backs out the changes. If you need to back out all changes, you can use the ROLL system service call to dynamically back out database changes.

Restarting your program

If you issue symbolic checkpoint calls (for batch and BMP programs), you can use the Extended Restart system service request (XRST) to restart your program after an abnormal termination.

The XRST call restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint request the program issued before terminating abnormally.

If you use basic checkpoint calls (for batch and BMP programs), you must provide the necessary code to restart the program from the latest checkpoint in the event that it terminates abnormally.

One way to restart the program from the latest checkpoint is to store repositioning data in an HDAM database. Your program writes a database record containing repositioning information to the HDAM database. It updates this record at intervals. When the program terminates, the database record is deleted. At the

completion of the XRST call, the I/O area always contains a checkpoint ID used by the restart. Normally, XRST will return the 8-byte symbolic checkpoint ID, followed by 4 blanks. If the 8-byte ID consists of all blanks, then XRST will return the 14-byte time-stamp ID. Also, check the status code in the PCB. The only successful status code for an XRST call is a row of blanks.

Related concepts:

"Developing JBP applications with the IMS Java dependent region resource adapter" on page 652

Data availability considerations for your CICS program

The data that a program needs to access may sometimes be unavailable. Use the following functions when data is not available.

Unavailability of a database

The conditions that make an entire database unavailable for both read and update are the following.

- A STOP command has been issued for the database.
- A DBRECOVERY (DBR) command has been issued for the database.
- DBRC authorization for the database has failed.

The conditions that make a database available for read but not for update are:

- A DBDUMP command has been issued for the database.
- The database access value is RD (read).

In a data-sharing environment, the command or error that created any of these conditions may have originated on the other system which is sharing data.

Whether a program is scheduled or whether an executing program can schedule a PSB when the database is unavailable depends on the type of program and the environment:

- A batch program

  IMS does not schedule a batch program when one of the databases that the program can access is not available.

  In a non-data sharing environment, DBRC authorization for a database may fail because the database is currently authorized to a DB/DC environment. In a data-sharing environment, a CICS or a DBCTL master terminal global command to recover a database or to dump a database may make the database unavailable to a batch program.

  The following conditions alone do not cause a batch program to fail during initialization:

  – A PCB refers to a HALDB.
  – The use of DBRC is suppressed.

  However, without DBRC, a database call using a PCB for a HALDB is not allowed. If the program is sensitive to unavailable data, such a call results in the status code BA; otherwise, such a call results in message DFS3303I, followed by ABENDU3303.
- An online or BMP program in the DBCTL environment.

  When a program executing in this environment attempts to schedule with a PSB containing one or more full-function databases that are unavailable, the scheduling is allowed. If the program does not attempt to access the unavailable

database, it can function normally. If it does attempt to access the database, the result is the same as when the database is available but some of the data in it is not available.

Unavailability of some data in a database

In addition to the situation where the entire database is unavailable, there are other situations where a limited amount of data is unavailable. One example is a failure situation involving data sharing where the IMS system knows which locks were held by a sharing IMS at the time the sharing IMS system failed. This IMS system continues to use the database but rejects access to the data that the failed IMS system held locked at the time of failure.

A batch program, an online program, or a BMP program can be operating in the DBCTL environment. If so, the online or BMP programs may have been scheduled when an entire database was not available. The following options apply to these programs when they attempt to access data and either the entire database is unavailable or only some of the data in the database is unavailable.

Programs executing in these environments have an option of being sensitive or insensitive to data unavailability.

- When the program is insensitive to data unavailability and attempts to access unavailable data, the program fails with a 3303 abend. For online programs, this is a pseudo-abend. For batch programs, it is a real abend. However, if the database is unavailable because dynamic allocation failed, a call results in an AI (unable to open) status code.
- When the program is sensitive to data unavailability and attempts to access unavailable data, IMS returns a status code indicating that it could not process the request. The program can then take the appropriate action. A facility exists for the program to then initiate the same action that IMS would have taken if the program had been insensitive to unavailable data.

The program issues the `INIT` call or `ACCEPT STATUS GROUP A` command to inform IMS that it is sensitive to unavailable data and can accept the status codes issued when the program attempts to access such data. The `INIT` request can also be used to determine data availability for each PCB in the PSB.

The SETS or SETU and ROLS functions

The `SETS` or `SETU` and `ROLS` requests allow an application to define multiple points at which to preserve the state of full-function databases.

The application can then return to these points at a later time. By issuing a `SETS` or `SETU` request before initiating a set of DL/I requests to perform a function, the program can later issue the `ROLS` request if it cannot complete the function due possibly to data unavailability.

`ROLS` allows the program to roll back its IMS activity to the state prior to the `SETS` or `SETU` call.

Restriction: `SETS` or `SETU` and `ROLS` only roll back the IMS updates. They do not roll back the updates made using CICS file control or transient data.

Additionally, you can use the `ROLS` call or command to undo all database update activity since the last checkpoint.

Use of STAE or ESTAE and SPIE in IMS batch programs

IMS uses STAE or ESTAE routines in the IMS batch regions to ensure that database logging and various resource cleanup functions are completed.

Two important aspects of the STAE or ESTAE facility are that:

- IMS relies on its STAE or ESTAE facility to ensure database integrity and resource control.
- The STAE or ESTAE facility is also available to the application program.

Because of these two factors, be sure you clearly understand the relationship between the program and the STAE or ESTAE facility.

Generally, do not use the STAE or ESTAE facility in your batch application program. However, if you believe that the STAE or ESTAE facility is required, you must observe the following basic rules:

- When the environment supports STAE or ESTAE processing, the application program STAE or ESTAE routines always get control before the IMS STAE or ESTAE routines. Therefore, you must ensure that the IMS STAE or ESTAE exit routines receive control by observing the following procedures in your application program:
  - Establish the STAE or ESTAE routine only once and always before the first DL/I call.
  - When using the STAE or ESTAE facility, the application program must not alter the IMS abend code.
  - Do not use the RETRY option when exiting from the STAE or ESTAE routine. Instead, return a CONTINUE-WITH-TERMINATION indicator at the end of the STAE or ESTAE processing. If your application program does specify the RETRY option, be aware that IMS STAE or ESTAE exit routines will not get control to perform cleanup. Therefore, system and database integrity may be compromised.
- The application program STAE/ESTAE exit routine must not issue DL/I calls because the original abend may have been caused by a problem between the application and IMS. This would result in recursive entry to STAE/ESTAE with potential loss of database integrity or in problems taking a checkpoint.

Dynamic allocation for IMS databases

Use the dynamic allocation function to specify the JCL information for IMS databases in a library instead of in the JCL of each batch job or in the JCL for DBCTL.

If you use dynamic allocation, do not include JCL DD statements for any database data sets that have been defined for dynamic allocation. Check with the database administrator (DBA) or comparable specialist at to determine which databases have been defined for dynamic allocation.

Related Reading: For more information on the definitions for dynamic allocation, see the DFSMDA macro in *IMS Version 11 System Definition*.

Chapter 5. Gathering requirements for database options

After designing hierarchies for the databases that your application will access, the DBA evaluates database options in terms of which options will best meet application requirements. Whether these options are used depends on the collected requirements of the applications. To design an efficient database, the DBA needs information about the individual applications.

Related concepts:

"Processing messages: Message Processing Programs" on page 40

Analyzing data access

The DBA chooses a type of database, based on how the majority of programs that use the database will access the data.

IMS databases are categorized according to the access method used. The following is a list of the types of databases that can be defined:

HDAM (Hierarchical Direct Access Method)
PHDAM (Partitioned Hierarchical Direct Access Method)
HIDAM (Hierarchical Indexed Direct Access Method)
PHIDAM (Partitioned Hierarchical Indexed Direct Access Method)
MSDB (Main Storage Database)
DEDB (Data Entry Database)
HSAM (Hierarchical Sequential Access Method)
HISAM (Hierarchical Indexed Sequential Access Method)
GSAM (Generalized Sequential Access Method)
SHSAM (Simple Hierarchical Sequential Access Method)
SHISAM (Simple Hierarchical Indexed Sequential Access Method)

Important: PHDAM and PHIDAM are the partitioned versions of the HDAM and HIDAM database types, respectively. The corresponding descriptions of the HDAM and HIDAM database types therefore apply to PHDAM and PHIDAM.

Some of the information that you can gather to help the DBA with this decision answers questions like the following:

- To access a database record, a program must first access the root of the record. How will each program access root segments?
  Directly
  Sequentially
  Both
- The segments within the database record are the dependents of the root segment. How will each program access the segments within each database record?
  Directly
  Sequentially
  Both

© Copyright IBM Corp. 1974, 2015

It is important to note the distinction between accessing a database record and accessing segments within the record. A program could access database records sequentially, but after the program is within a record, the program might access the segments directly. These are different, and can influence the choice of access method.

- To what extent will the program update the database?

  By adding new database records?

  By adding new segments to existing database records?

  By deleting segments or database records?

Again, note the difference between updating a database record and updating a segment within the database record.

Direct access

The advantage of direct access processing is that you can get good results for both direct and sequential processing. Direct access means that by using a randomizing routine or an index, IMS can find any database record that you want, regardless of the sequence of database records in the database.

IMS full function has four direct access methods.

- HDAM and PHDAM process data directly by using a randomizing routine to store and locate root segments.
- HIDAM and PHIDAM use an index to help them provide direct processing of root segments.

The direct access methods use pointers to maintain the hierarchic relationships between segments of a database record. By following pointers, IMS can access a path of segments without passing through all the segments in the preceding paths.

Some of the requirements that direct access satisfies are:

- Fast direct processing of roots using an index or a randomizing routine
- Sequential processing of database records with HIDAM and PHIDAM using the index
- Fast access to a path of segments using pointers

In addition, when you delete data from a direct-access database, the new space is available almost immediately. This gives you efficient space utilization; therefore, reorganization of the database is often unnecessary. Direct access methods internally maintain their own pointers and addresses.

A disadvantage of direct access is that you have a larger IMS overhead because of the pointers. But if direct access fulfills your data access requirements, it is more efficient than using a sequential access method.

Primarily direct processing: HDAM

HDAM is efficient for a database that is usually accessed directly but sometimes sequentially. HDAM uses a randomizing routine to locate its root segments and then chains dependent segments together according to the pointer options chosen. The z/OS access methods that HDAM can use are Virtual Storage Access Method (VSAM) and Overflow Storage Access Method (OSAM).

Important: PHDAM is the partitioned version of the HDAM database type. The corresponding descriptions of the HDAM database type therefore apply to PHDAM.

The requirements that HDAM satisfies are:

- Direct access of roots by root keys because HDAM uses a randomizing routine to locate root segments
- Direct access of paths of dependents
- Adding new database records and new segments because the new data goes into the nearest available space
- Deleting database records and segments because the space created by a deletion can be used by any new segment

HDAM characteristics

An HDAM database:

- Can store root segments anywhere. Root segments do not need to be in sequence because the randomizing routine locates them.
- Uses a randomizing routine to locate the relative block number and root anchor point (RAP) within the block that points to the root segment.
- Accesses the RAPs from which the roots are chained in physical sequence. Then the root segments that are chained from the root anchors are returned. Therefore, sequential retrieval of root segments from HDAM is not based on the results of the randomizing routine and is not in key sequence unless the randomizing routine put them into key sequence.
- May not give the desired result for some calls unless the randomizing module causes the physical sequence of root segments to be in the key sequence. For example, a `GU` call for a root segment that is qualified as less than or equal to a root key value would scan in physical sequence for the first RAP of the first block. This may result in a not-found condition, even though segments meeting the qualification do exist.

For dependent segments, an HDAM database:

- Can store them anywhere
- Chains all segments of one database record together with pointers

An Overview of how HDAM works

This topic contains Diagnosis, Modification, and Tuning information.

When a database record is stored in an HDAM database, HDAM keeps one or more RAPs at the beginning of each physical block. The RAP points to a root segment. HDAM also keeps a pointer at the beginning of each physical block that points to any free space in the block. When you insert a segment, HDAM uses this pointer to locate free space in the physical block. To locate a root segment in an HDAM database, you give HDAM the root key. The randomizing routine gives it the relative physical block number and the RAP that points to the root segment. The specified RAP number gives HDAM the location of the root within a physical block.

Although HDAM can place roots and dependents anywhere in the database, it is better to choose HDAM options that keep roots and dependents close together.

HDAM performance depends largely on the randomizing routine you use. Performance can be very good, but it also depends on other factors such as:

- The block size you use
- The number of RAPs per block
- The pattern for chaining together different segments. You can chain segments of a database record in two ways:
  - In hierarchic sequence, starting with the root
  - In parent-to-dependent sequence, with parents having pointers to each of their paths of dependents

To use HDAM for sequential access of database records by root key, you need to use a secondary index or a randomizing routine that stores roots in physical key sequence.

Direct and sequential processing: HIDAM

HIDAM is the access method that is most efficient for an approximately equal amount of direct and sequential processing.

Important: PHIDAM is the partitioned version of the HIDAM database type. The corresponding descriptions of the HIDAM database type therefore apply to PHIDAM.

The z/OS access methods it can use are VSAM and OSAM. The specific requirements that HIDAM satisfies are:

- Direct and sequential access of records by their root keys
- Direct access of paths of dependents
- Adding new database records and new segments because the new data goes into the nearest available space
- Deleting database records and segments because the space created by a deletion can be used by any new segment

HIDAM can satisfy most processing requirements that involve an even mixture of direct and sequential processing. However, HIDAM is not very efficient with sequential access of dependents.

HIDAM characteristics

For root segments, a HIDAM database:

- Initially loads them in key sequence
- Can store new root segments wherever space is available
- Uses an index to locate a root that you request and identify by supplying the root's key value

For dependent segments, a HIDAM database:

- Can store segments anywhere, preferably fairly close together
- Chains all segments of a database record together with pointers

An overview of how HIDAM works

This topic contains Diagnosis, Modification, and Tuning information.

HIDAM uses two databases. The primary database holds the data. An index database contains entries for all of the root segments in order by their key fields. For each key entry, the index database contains the address of that root segment in the primary database.

When you access a root, you supply the key to the root. HIDAM looks up the key in the index to find the address of the root and then goes to the primary database to find the root.

HIDAM chains dependent segments together so that when you access a dependent segment, HIDAM uses the pointer in one segment to locate the next segment in the hierarchy.

When you process database records directly, HIDAM locates the root through the index and then locates the segments from the root. HIDAM locates dependents through pointers.

If you plan to process database records sequentially, you can specify special pointers in the DBD for the database so that IMS does not need to go to the index to locate the next root segment. These pointers chain the roots together. If you do not chain roots together, HIDAM always goes to the index to locate a root segment. When you process database records sequentially, HIDAM accesses roots in key sequence in the index. This only applies to sequential processing; if you want to access a root segment directly, HIDAM uses the index, and not pointers in other root segments, to find the root segment you have requested.

Main storage database: MSDB

Use MSDBs to store the most frequently-accessed data. MSDBs are suitable for applications such as general ledger applications in the banking industry.

Recommendation: Use DEDBs instead of MSDBs when you develop new Fast Path databases. Terminal-related MSDBs and non-terminal-related MSDBs with terminal-related keys are no longer supported. Although non-terminal-related MSDBs with non-terminal-related-keys are still supported, you should consider converting any existing MSDBs to DEDBs. You can use the MSDB-to-DEDB Conversion utility.

MSDB characteristics

MSDBs reside in virtual storage, enabling application programs to avoid the I/O activity that is required to access them. The two kinds of MSDBs are terminal-related and non-terminal-related.

In a terminal-related MSDB, each segment is owned by one terminal, and each terminal owns only one segment. One use for this type of MSDB is an application in which each segment contains data associated with a logical terminal. In this type of application, the program can read the data (perhaps for reporting purposes), but cannot update it. A non-terminal-related MSDB stores data that is needed by many users during the same time period. It can be updated and read from all terminals (for example, a real time inventory control application, where reduction of inventory can be noted from many cash registers).

An overview of how MSDBs work

This topic contains Diagnosis, Modification, and Tuning information.

MSDB segments are stored as root segments only. Only one type of pointer, the forward chain pointer, is used. This pointer connects the segment records in the database.

Data entry database: DEDB

DEDBs are designed to provide access to and efficient storage for large volumes of data. The primary requirement a DEDB satisfies is a high level of data availability.

DEDB characteristics

DEDBs are hierarchic databases that can have as many as 15 hierarchic levels, and as many as 127 segment types. They can contain both direct and sequential dependent segments. Because the sequential dependent segments are stored in chronological order as they are committed to the database, they are useful in journaling applications.

DEDBs support a subset of functions and options that are available for a HIDAM or HDAM database. For example, a DEDB does not support indexed access (neither primary index nor secondary index), or logically related segments.

An overview of how DEDBs work

This topic contains Diagnosis, Modification, and Tuning information.

A DEDB can be partitioned into multiple areas, with each area containing a different collection of database records. The data in a DEDB area is stored in a VSAM data set. Root segments are stored in the root-addressable part of an area, with direct dependents stored close to the roots for fast access. Direct dependents that cannot be stored close to their roots are stored in the independent overflow portion of the area. Sequential dependents are stored in the sequential dependent portion at the end of the area so that they can be quickly inserted. Each area data set can have up to seven copies, making the data easily available to application programs.

Sequential access

When you use a sequential access method, the segments in the database are stored in hierarchic sequence, one after another, with no pointers.

IMS full-function has two sequential access methods. Like the direct access methods, one has an index and the other does not:

- HSAM only processes root segments and dependent segments sequentially.
- HISAM processes data sequentially but has an index so that you can access records directly. HISAM is primarily for sequentially processing dependents, and directly processing database records.

Some of the general requirements that sequential access satisfies are:

- Fast sequential processing
- Direct processing of database records with HISAM
- Small IMS overhead on storage because sequential access methods relate segments by adjacency rather than with pointers

The three disadvantages of using sequential access methods are:

- Sequential access methods give slower access to the right-most segments in the hierarchy, because HSAM and HISAM must read through all other segments to get to them.
- HISAM requires frequent reorganization to reclaim space from deleted segments and to keep the logical records of a database record physically adjoined.
- You cannot update HSAM databases. You must create a new database to change any of the data.

Sequential processing only: HSAM

HSAM is a hierarchic access method that can handle only sequential processing. You can retrieve data from HSAM databases, but you cannot update any of the data. The z/OS access methods that HSAM can use are QSAM and BSAM.

HSAM is ideal for the following situations:

- You are using the database to collect (but not update) data or statistics.
- You only plan to process the data sequentially.

HSAM characteristics

HSAM stores database records in the sequence in which you submit them. You can only process records and dependent segments sequentially, which means the order in which you have loaded them. HSAM stores dependent segments in hierarchic sequence.

An overview of how HSAM works

This topic contains Diagnosis, Modification, and Tuning information.

HSAM databases are very simple databases. The data is stored in hierarchic sequence, one segment after the other, and no pointers or indexes are used.

Primarily sequential processing: HISAM

HISAM is an access method that stores segments in hierarchic sequence with an index to locate root segments. It also has an overflow data set. Store segments in a logical record until you reach the end of the logical record. When you run out of space on the logical record, but you still have more segments belonging to the database record, you store the remaining segments in an overflow data set. The access methods that HISAM can use are VSAM and OSAM.

HISAM is well-suited for:

- Direct access of record by root keys
- Sequential access of records
- Sequential access of dependent segments

The situations in which your processing has some of these characteristics but where HISAM is not necessarily a good choice, occur when:

- You must access dependents directly.
- You have a high number of inserts and deletes.
- Many of the database records exceed average size and must use the overflow data set. The segments that overflow into the overflow data set require additional I/O.

HISAM characteristics

For database records, HISAM databases:

- Store records in key sequence
- Can locate a particular record with a key value by using the index

For dependent segments, HISAM databases:

- Start each HISAM database record in a new logical record in the primary data set
- Store the remaining segments in one or more logical records in the overflow data set if the database record does not fit in the primary data set

An overview of how HISAM works

This topic contains Diagnosis, Modification, and Tuning information.

HISAM does not immediately reuse space. When you insert a new segment, HISAM databases shift data to make room for the new segment, and this leaves unused space after deletions. HISAM space is reclaimed when you reorganize a HISAM database.

Accessing z/OS files through IMS: GSAM

GSAM enables IMS batch application programs and BMPs to access a sequential z/OS data set as a simple database. The z/OS access methods that GSAM can use are BSAM and VSAM. A GSAM database is a z/OS data set record that is defined as a database record. The record is handled as one unit; it contains no segments or fields and the structure is not hierarchic. GSAM databases can be accessed by z/OS, IMS, and CICS.

In a CICS environment, an application program can access a GSAM database from either a Call DL/I (or EXEC DLI) batch or batch-oriented BMP program. A CICS application cannot, however, use EXEC DLI to process GSAM databases; it must use IMS calls.

You commonly use GSAM to send input to and receive output from batch-oriented BMPs or batch programs. To process a GSAM database, an application program issues calls similar to the ones it issues to process a full-function database. The program can read data sequentially from a GSAM database, and it can send output to a GSAM database.

GSAM is a sequential access method. You can only add records to an output database sequentially.

Accessing IMS data through z/OS: SHSAM and SHISAM

Two database access methods give you simple hierarchic databases that z/OS can use as data sets, SHSAM and SHISAM.

These access methods can be particularly helpful when you are converting data from z/OS files to an IMS database. SHISAM is indexed and SHSAM is not.

When you use these access methods, you define an entire database record as one segment. The segment does not contain any IMS control information or pointers; the data format is the same as it is in z/OS data sets. The z/OS access methods that SHSAM can use are BSAM and QSAM. SHISAM uses VSAM.

SHSAM and SHISAM databases can be accessed by z/OS access methods without IMS, which is useful during transitions.

Understanding how data structure conflicts are resolved

The order in which application programs need to process fields and segments within hierarchies is frequently not the same for each application. When the DBA finds a conflict in the way that two or more programs need to access the data, three options are available to solve these problems. Each of the following options solves a different kind of conflict.

- When an application program does not need access to all the fields in a segment, or if the program needs to access them in a different order, the DBA can use **field level sensitivity** for that program. *Field-level sensitivity* makes it possible for an application program to access only a subset of the fields that a segment contains, or for an application program to process a segment's fields in an order that is different from their order in the segment.
- When an application program needs to access a particular segment by a field other than the segment's key field, the DBA can use a **secondary index** for that database.
- When the application program needs to relate segments from different hierarchies, the DBA can use **logical relationships**. Using logical relationships can give the application program a logical hierarchy that includes segments from several hierarchies.

Related concepts:

"Determining mappings" on page 28

Using different fields: field-level sensitivity

Field-level sensitivity applies the same kind of security for fields within a segment that segment sensitivity does for segments within a hierarchy: An application program can access only those fields within a segment, and those segments within a hierarchy to which it is sensitive.

Field-level sensitivity also makes it possible for an application program to use a subset of the fields that make up a segment, or to use all the fields in the segment but in a different order. If a segment contains fields that the application program does not need to process, using field-level sensitivity enables the program not to process them.

Example of field-level sensitivity

Suppose that a segment containing data about an employee contains the fields shown in the following table. These fields are:

- Employee number: EMPNO
- Employee name: EMPNAME
- Birthdate: BIRTHDAY
- Salary: SALARY
- Address: ADDRESS

Table 24. Physical employee segment

EMPNO	EMPNAME	BIRTHDAY	SALARY	ADDRESS

A program that printed mailing labels for employees' checks each week would not need all the data in the segment. If the DBA decided to use field-level sensitivity for that application, the program would receive only the fields it needed in its I/O area. The I/O area would contain the EMPNAME and ADDRESS fields. The following table shows what the program's I/O area would contain.

Table 25. Employee segment with field-level sensitivity

EMPNAME	ADDRESS

Field-level sensitivity makes it possible for a program to receive a subset of the fields that make up a segment, the same fields but in a different order, or both.

Another situation in which field-level sensitivity is very useful is when new uses of the database involve adding new fields of data to an existing segment. In this situation, you want to avoid re-coding programs that use the current segment. By using field-level sensitivity, the old programs can see only the fields that were in the original segment. The new program can see both the old and the new fields.

Specifying field-level sensitivity

You specify field-level sensitivity in the PSB for the application program by using a sensitive field (SENFLD) statement for each field to which you want the application program to be sensitive.

Related reference:

SENFLD statement (System Utilities)

Resolving processing conflicts in a hierarchy: secondary indexing

Sometimes a database hierarchy does not meet all the processing requirements of the application programs that will process it.

Secondary indexing can be used to solve two kinds of processing conflicts:

- When an application program needs to retrieve a segment in a sequence other than the one that has been defined by the segment's key field
- When an application program needs to retrieve a segment based on a condition that is found in a dependent of that segment

To understand these conflicts and how secondary indexing can resolve them, consider the examples of two application programs that process the patient hierarchy, shown in the following figure. Three segment types in this hierarchy are:

- PATIENT contains three fields: the patient's identification number, name, and address. The patient number field is the key field.
- ILLNESS contains two fields: the date of the illness and the name of the illness. The date of the illness is the key field.
- TREATMNT contains four fields: the date the medication was given; the name of the medication; the quantity of the medication that was given; and the name of the doctor who prescribed the medication. The date that the medication was given is the key field.

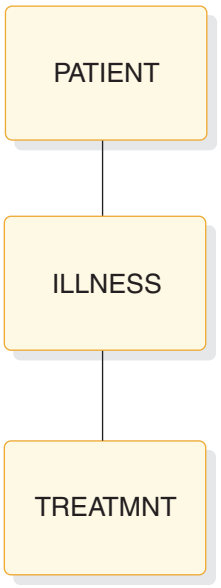


Figure 16. Patient hierarchy

Retrieving segments based on a different key

When an application program retrieves a segment from the database, the program identifies the segment by the segment's key field. But sometimes an application program needs to retrieve a segment in a sequence other than the one that has been defined by the segment's key field. Secondary indexing makes this possible.

Note: A new database type, the Partitioned Secondary Index (PSINDEX), is supported by the High Availability Large Database (HALDB). PSINDEX is the partitioned version of the secondary index database type. The corresponding descriptions of the secondary index database type therefore apply to PSINDEX.

For example, suppose you have an online application program that processes requests about whether an individual has ever been to the clinic. If you are not sure whether the person has ever been to the clinic, you will not be able to supply the identification number for the person. But the key field of the PATIENT segment is the patient's identification number.

Segment occurrences of a segment type (for example, the segments for each of the patients) are stored in a database in order of their keys (in this case, by their patient identification numbers). If you issue a request for a PATIENT segment and identify the segment you want by the patient's name instead of the patient's identification number, IMS must search through all of the PATIENT segments to find the PATIENT segment you have requested. IMS does not know where a particular PATIENT segment is just by having the patient's name.

To make it possible for this application program to retrieve PATIENT segments in the sequence of patients' names (rather than in the sequence of patients' identification numbers), you can index the PATIENT segment on the patient name field and store the index entries in a separate database. The separate database is called a *secondary index database*.

Then, if you indicate to IMS that it is to process the PATIENT segments in the patient hierarchy in the sequence of the index entries in the secondary index

database, IMS can locate a PATIENT segment if you supply the patient's name. IMS goes directly to the secondary index and locates the PATIENT index entry with the name you have supplied; the PATIENT index entries are in alphabetical order of the patient names. The index entry is a pointer to the PATIENT segment in the patient hierarchy. IMS can determine whether a PATIENT segment for the name you have supplied exists, and then it can return the segment to the application program if the segment exists. If the requested segment does not exist, IMS indicates this to the application program by returning a not-found status code.

Related reading: For more information on HALDB, see *IMS Version 11 Database Administration*.

Three terms involved in secondary indexing are:

Pointer segment
The index entry in the secondary index database that IMS uses to find the segment you have requested. In the previous example, the pointer segment is the index entry in the secondary index database that points to the PATIENT segment in the patient hierarchy.

Source segment
The segment that contains the field that you are indexing. In the previous example, the source segment is the PATIENT segment in the patient hierarchy, because you are indexing on the name field in the PATIENT segment.

Target segment
The segment in the database that you are processing to which the secondary index points; it is the segment that you want to retrieve.

In the previous example, the target segment and the source segment are the same segment—the PATIENT segment in the patient hierarchy. When the source segment and the target segment are different segments, secondary indexing solves the processing conflict.

The PATIENT segment that IMS returns to the application program's I/O area looks the same as it would if secondary indexing had not been used.

The key feedback area is different. When IMS retrieves a segment without using a secondary index, IMS places the concatenated key of the retrieved segment in the key feedback area. The concatenated key contains all the keys of the segment's parents, in order of their positions in the hierarchy. The key of the root segment is first, followed by the key of the segment on the second level in the hierarchy, then the third, and so on—with the key of the retrieved segment last.

But when you retrieve a segment from an indexed database, the contents of the key feedback area after the request are a little different. Instead of placing the key of the root segment in the left-most bytes of the key feedback area, DL/I places the key of the pointer segment there. Note that the term “key of the pointer segment,” as used here, refers to the key as perceived by the application program—that is, the key does not include subsequence fields.

For example, suppose index segment A shown in the following figure is indexed on a field in segment C. Segment A is the target segment, and segment C is the source segment.

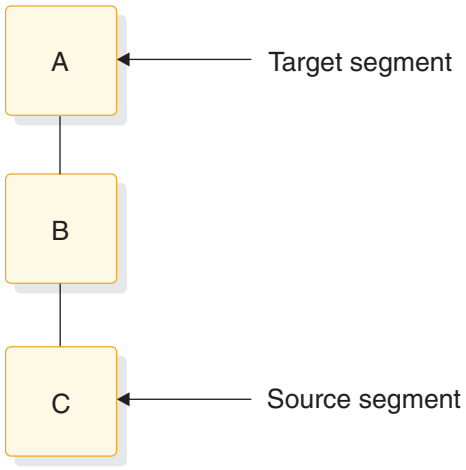


Figure 17. Indexing a root segment

When you use the secondary index to retrieve one of the segments in this hierarchy, the key feedback area contains one of the following:

- If you retrieve segment A, the key feedback area contains the key of the pointer segment from the secondary index.
- If you retrieve segment B, the key feedback area contains the key of the pointer segment, concatenated with the key of segment B.
- If you retrieve segment C, the key of the pointer segment, the key of segment B, and the key of segment C are concatenated in the key feedback area.

Although this example creates a secondary index for the root segment, you can index dependent segments as well. If you do this, you create an inverted structure: the segment you index becomes the root segment, and its parent becomes a dependent.

For example, suppose you index segment B on a field in segment C. In this case, segment B is the target segment, and segment C is the source field. The following figure shows the physical database structure and the structure that is created by the secondary index.

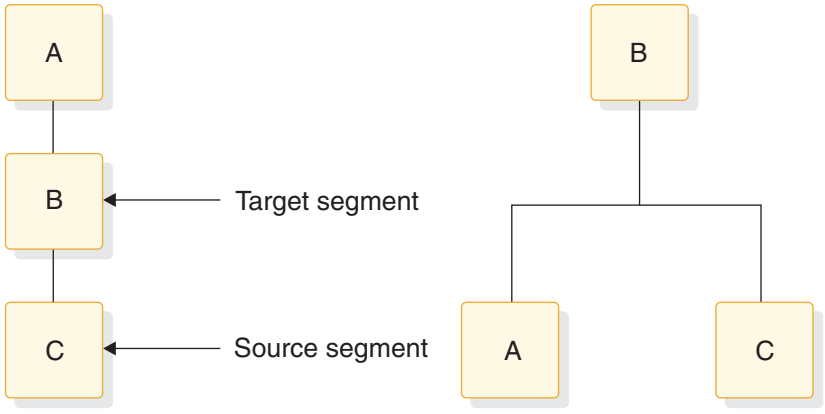


Figure 18. Indexing a dependent segment

When you retrieve the segments in the secondary index data structure on the right, IMS returns the following to the key feedback area:

- If you retrieve segment B, the key feedback area contains the key of the pointer segment in the secondary index database.
- If you retrieve segment A, the key feedback area contains the key of the pointer segment, concatenated with the key of segment A.
- If you retrieve segment C, the key feedback area contains the key of the pointer segment, concatenated with the key of segment C.

Retrieving segments based on the qualification of a dependent segment

Sometimes an application program needs to retrieve a segment, but only if one of the dependents of the segment meet certain qualifications.

For example, suppose that the medical clinic wants to print a monthly report of the patients who have visited the clinic during that month. If the application program that processes this request does not use a secondary index, the program has to retrieve each PATIENT segment, and then retrieve the ILLNESS segment for each PATIENT segment. The program tests the date in the ILLNESS segment to determine whether the patient has visited the clinic during the current month, and prints the patient's name if the answer is yes. The program continues retrieving PATIENT segments and ILLNESS segments until it has retrieved all the PATIENT segments.

But with a secondary index, you can make the processing of the program simpler. To do this, you index the PATIENT segment on the date field in the ILLNESS segment. When you define the PATIENT segment in the DBD, you give IMS the name of the field on which you are indexing the PATIENT segment, and the name of the segment that contains the index field. The application program can then request a PATIENT segment and qualify the request with the date in the ILLNESS segment. The PATIENT segment that is returned to the application program looks just as it would if you were not using a secondary index.

In this example, the PATIENT segment is the target segment; it is the segment that you want to retrieve. The ILLNESS segment is the source segment; it contains the information that you want to use to qualify your request for PATIENT segments. The index segment in the secondary database is the pointer segment. It points to the PATIENT segments.

Creating a new hierarchy: logical relationships

When an application program needs to associate segments from different hierarchies, logical relationships can make that possible.

Logical relationships can solve the following conflicts:

- When two application programs need to process the same segment, but they need to access the segment through different hierarchies
- When a segment's parent in one application program's hierarchy acts as that segment's child in another application program

Accessing a segment through different paths

Sometimes an application program needs to process the data in a different order than the way it is arranged in the hierarchy.

For example, an application program that processes data in a purchasing database also requires access to a segment in a patient database:

- Program A processes information in the patient database about the patients at a medical clinic: the patients' illnesses and their treatments.
- Program B is an inventory program that processes information in the purchasing database about the medications that the clinic uses: the item, the vendor, information about each shipment, and information about when and under what circumstances each medication is given.

The following figure shows the hierarchies that Program A and Program B require for their processing. Their processing requirements conflict: they both need to have access to the information that is contained in the TREATMNT segment in the patient database. This information is:

- The date that a particular medication was given
- The name of the medication
- The quantity of the medication given
- The doctor that prescribed the medication

To Program B this is not information about a patient's treatment; it is information about the disbursement of a medication. To the purchasing database, this is the disbursement segment (DISBURSE).

The following figure shows the hierarchies for Program A and Program B. Program A needs the PATIENT segment, the ILLNESS segment, and the TREATMNT segment. Program B needs the ITEM segment, the VENDOR segment, the SHIPMENT segment, and the DISBURSE segment. The TREATMNT segment and the DISBURSE segment contain the same information.

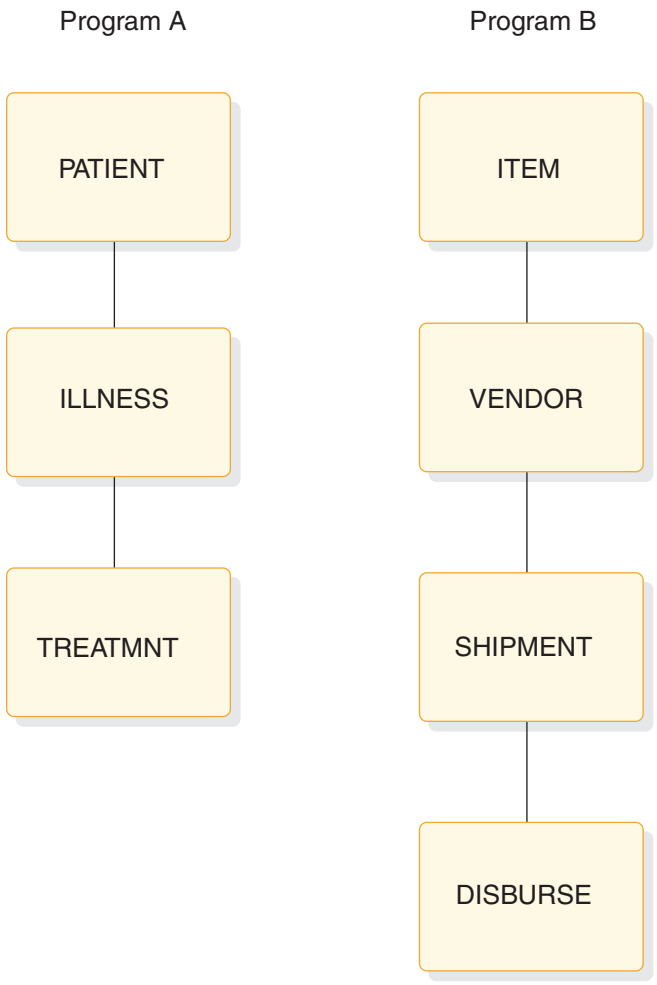


Figure 19. Patient and inventory hierarchies

Instead of storing this information in both hierarchies, you can use a logical relationship. A logical relationship solves the problem by storing a pointer from where the segment is needed in one hierarchy to where the segment exists in the other hierarchy. In this case, you can have a pointer in the DISBURSE segment to the TREATMNT segment in the medical database. When IMS receives a request for information in a DISBURSE segment in the purchasing database, IMS goes to the TREATMNT segment in the medical database that is pointed to by the DISBURSE segment. The following figure shows the physical hierarchy that Program A would process and the logical hierarchy that Program B would process. DISBURSE is a pointer segment to the TREATMNT segment in Program A's hierarchy.

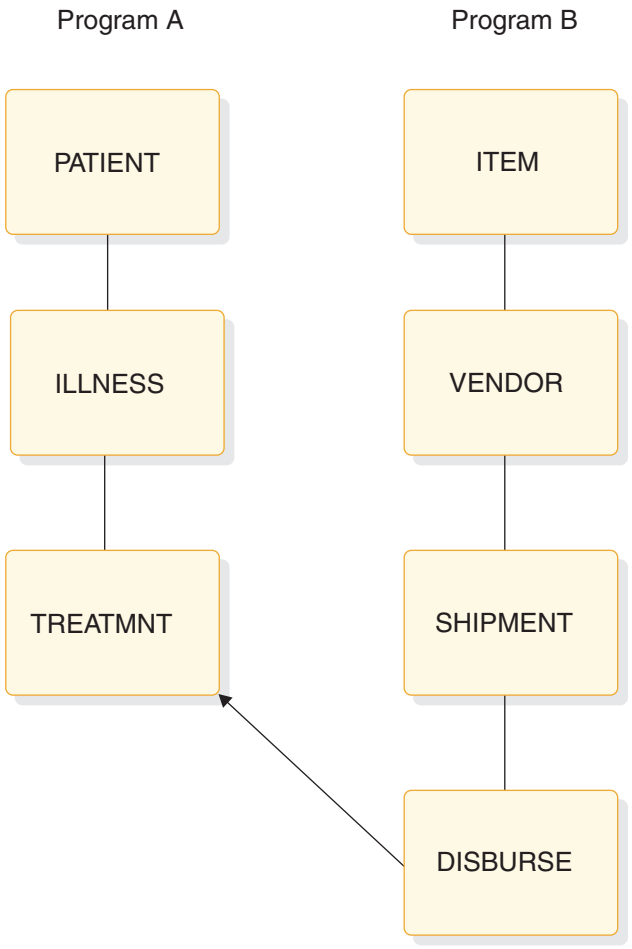


Figure 20. Logical relationships example

To define a logical relationship between segments in different hierarchies, you use a logical DBD. A logical DBD defines a hierarchy that does not exist in storage, but can be processed as though it does. Program B would use the logical structure shown in the previous figure as though it were a physical structure.

Inverting a parent-child relationship

Another type of conflict that logical relationships can resolve occurs when a segment's parent in one application program acts as that segment's child in another application program.

- The inventory program, Program B, needs to process information about medications using the medication as the root segment.
- A purchasing application program, Program C, processes information about which vendors have sold which medications. Program C needs to process this information using the vendor as the root segment.

The following figure shows the hierarchies for each of these application programs.

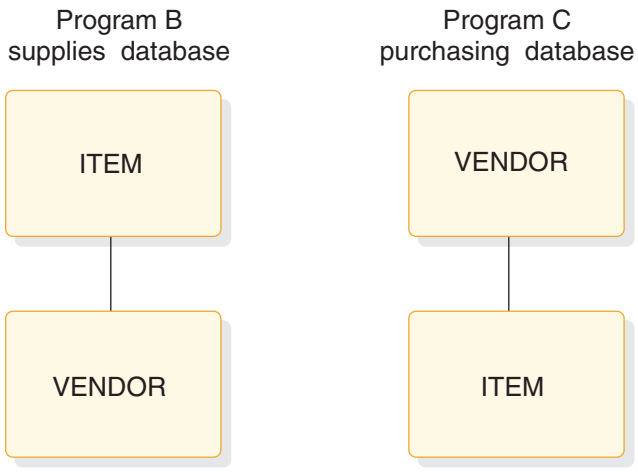


Figure 21. Supplies and purchasing hierarchies

Logical relationships can solve this problem by using pointers. Using pointers in this example would mean that the ITEM segment in the purchasing database would contain a pointer to the actual data stored in the ITEM segment in the supplies database. The VENDOR segment, however, would actually be stored in the purchasing database. The VENDOR segment in the supplies database would point to the VENDOR segment that is stored in the purchasing database.

The following figure shows the hierarchies of these two programs.

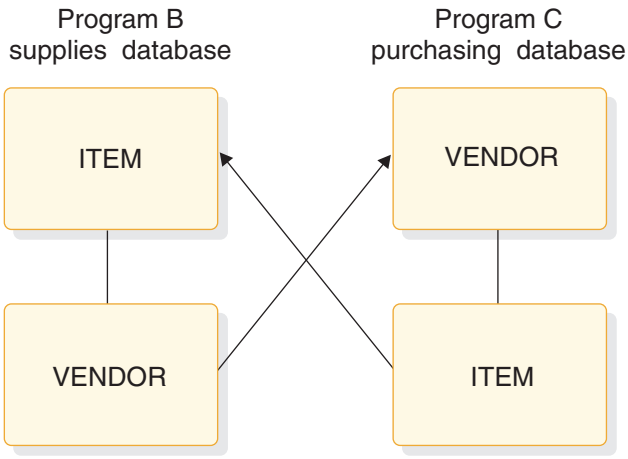


Figure 22. Program B and program C hierarchies

If you did not use logical relationships in this situation, you would:

- Keep the same data in both paths, which means that you would be keeping redundant data.
- Have the same disadvantages as separate files of data:
  - You would need to update multiple segments each time one piece of data changed.
  - You would need more storage.

Providing data security

You can control the security of data accessed by your IMS application programs through data sensitivity and processing options.

Data sensitivity
Controls what data a particular program can access.

Processing options
Controls how a particular program can process data that it can access.

Providing data availability

Specifying segment sensitivity and processing options also affects data availability. You should set the specifications so that the PCBs request the fewest SENSEGS and limit the possible processing options. With data availability, a program can continue to access and update segments in the database successfully, even though some parts of the database are unavailable.

The SENSEG statement defines a segment type in the database to which the application program is sensitive. A separate SENSEG statement must exist for each segment type. The segments can physically exist in one database or they can be derived from several physical databases. If an application program is sensitive to a segment that is below the root segment, it must also be sensitive to all segments in the path from the root segment to the sensitive segment.

Related Reading: For more information on using field-level sensitivity for data security and using the SENSEG statement to limit the scope of the PCBs, see *IMS Version 11 Database Administration*.

Related concepts:

"An overview of application design" on page 15

Keeping a program from accessing the data: data sensitivity

An IMS program can only access data to which it is sensitive.

You can control the data to which your program is sensitive on three levels:

- **Segment sensitivity** can prevent an application program from accessing all the segments in a particular hierarchy. Segment sensitivity tells IMS which segments in a hierarchy the program is allowed to access.
- **Field-level sensitivity** can keep a program from accessing all the fields that make up a particular segment. Field-level sensitivity tells IMS which fields within a particular segment a program is allowed to access.
- **Key sensitivity** means that the program can access segments below a particular segment, but it cannot access the particular segment. IMS returns only the key of this type of segment to the program.

You define each of these levels of sensitivity in the PSB for the application program. Key sensitivity is defined in the processing option for the segment. Processing options indicate to IMS exactly what a particular program may or may not do to the data. You specify a processing option for each hierarchy that the application program processes; you do this in the DB PCB that represents each hierarchy. You can specify one processing option for all the segments in the hierarchy, or you can specify different processing options for different segments within the hierarchy.

Segment sensitivity and field-level sensitivity are defined using special statements in the PSB.

Segment sensitivity

You define what segments an application program is sensitive to in the DB PCB for the hierarchy that contains those segments.

For example, suppose that the patient hierarchy shown in the following figures. The patient hierarchy is like a subset of the medical database.

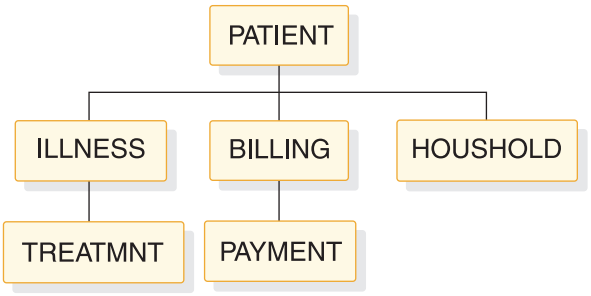


Figure 23. Medical database hierarchy

PATIENT is the root segment and the parent of the three segments below it: ILLNESS, BILLING, and HOUSHOLD. Below ILLNESS is TREATMNT. Below BILLING is PAYMENT.

To make it possible for an application program to view only the segments PATIENT, ILLNESS, and TREATMNT from the medical database, you specify in the DB PCB that the hierarchy you are defining has these three segment types, and that they are from the medical database. You define the database hierarchy in the DBD; you define the application program's view of the database hierarchy in the DB PCB.

Field-level sensitivity

In addition to providing data independence for an application program, field-level sensitivity can also act as a security mechanism for the data that the program uses.

If a program needs to access some of the fields in a segment, but one or two of the fields that the program does not need to access are confidential, you can use field-level sensitivity. If you define that segment for the application program as containing only the fields that are not confidential, you prevent the program from accessing the confidential fields. Field-level sensitivity acts as a mask for the fields to which you want to restrict access.

Key sensitivity

To access a segment, an application program must be sensitive to all segments at a higher level in the segment's path. In other words, in the following figure, a program must be sensitive to segment B in order to access segment C.

For example, suppose that an application program needs segment C to do its processing. But if segment B contains confidential information (such as an employee's salary), the program is not able to access that segment. Using key

sensitivity lets you withhold segment B from the application program while giving the program access to the dependents of segment B.

When a sensitive segment statement has a processing option of K specified for it, the program cannot access that segment, but the program can pass beyond that segment to access the segment's dependents. When the program does access the segment's dependents, IMS does not return that segment; IMS returns only the segment's key with the keys of the other segments that are accessed.

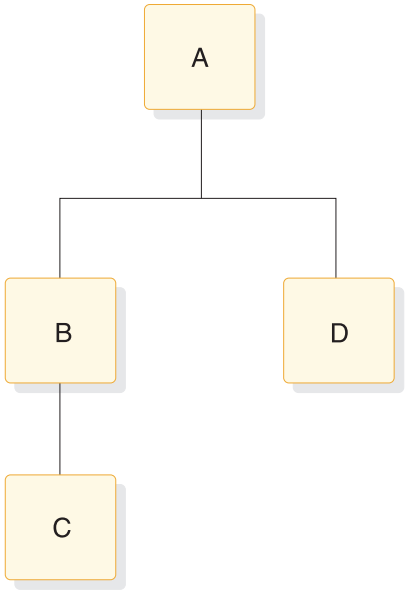


Figure 24. Sample hierarchy for key sensitivity example

Preventing a program from updating data: processing options

During PCB generation, you can use five options of the PROCOPT parameter (in the DATABASE macro) to indicate to IMS whether your program can read segments in the hierarchy, or whether it can also update segments.

From most restrictive to least restrictive, these options are:

G Your program can read segments.

R Your program can read and replace segments.

I Your program can insert segments.

D Your program can read and delete segments.

A Your program can perform all the processing options. It is equivalent to specifying G, R, I, and D.

Related Reading: For a thorough description of the processing options see, *IMS Version 11 System Utilities*.

Processing options provide data security because they limit what a program can do to the hierarchy or to a particular segment. Specifying only the processing options the program requires ensures that the program cannot update any data it is not supposed to. For example, if a program does not need to delete segments from a database, the D option need not be specified.

When an application program retrieves a segment and has any of the just-described processing options, IMS locks the database record for that application. If PROCOPT=G is specified, other programs with the option can concurrently access the database record. If an update processing option (R, I, D, or A) is specified, no other program can concurrently access the same database record. If no updates are performed, the lock is released when the application moves to another database record or, in the case of HDAM, to another anchor point.

The following locking protocol allows IMS to make this determination. If the root segment is updated, the root lock is held at update level until commit. If a dependent segment is updated, it is locked at update level. When exiting the database record, the root segment is demoted to read level. When a program enters the database record and obtains the lock at either read or update level, the lock manager provides feedback indicating whether or not another program has the lock at read level. This determines if dependent segments will be locked when they are accessed. For HISAM, the primary logical record is treated as the root, and the overflow logical records are treated as dependent segments.

When using block-level or database-level data sharing for online and batch programs, you can use additional processing options.

Related Reading:

- For a special case involving HISAM delete byte with parameter ERASE=YES see, *IMS Version 11 Database Administration*.
- For more information on database and block-level data sharing, see *IMS Version 11 System Administration*.

E option

With the E option, your program has exclusive access to the hierarchy or to the segment you use it with. The E option is used in conjunction with the options G, I, D, R, and A. While the E program is running, other programs cannot access that data, but may be able to access segments that are not in the E program's PCB. No dynamic enqueue by program isolation is done, but dynamic logging of database updates will be done.

GO option

When your program retrieves a segment with the GO option, IMS does not lock the segment. While the read without integrity program reads the segment, it remains available to other programs. This is because your program can only read the data (termed *read-only*); it is not allowed to update the database. No dynamic enqueue is done by program isolation for calls against this database. Serialization between the program with PROCOPT=GO and any other update program does not occur; updates to the same data occur simultaneously.

If a segment has been deleted and another segment of the same type has been inserted in the same location, the segment data and all subsequent data that is returned to the application may be from a different database record.

A read-without-integrity program can also retrieve a segment even if another program is updating the segment. This means that the program need not wait for segments that other programs are accessing. If a read-without-integrity program reads data that is being updated by another program, and that program terminates

abnormally before reaching the next commit point, the updated segments might contain invalid pointers. If an invalid pointer is detected, the read-without-integrity program terminates abnormally, unless the N or T options were specified with GO. Pointers are updated during insert, delete and backout functions.

N option

When you use the N option with GO to access a full-function database or a DEDB, and the segment you are retrieving contains an invalid pointer, IMS returns a GG status code to your program. Your program can then terminate processing, continue processing by reading a different segment, or access the data using a different path. The N option must be specified as PROCOPT=GON, GON, or GONP.

T option

When you use the T option with GO and the segment you are retrieving contains an invalid pointer, the response from an application program depends on whether the program is accessing a full-function or Fast Path database.

For calls to full-function databases, the T option causes DL/I to automatically retry the operation. You can retrieve the updated segment, but only if the updating program has reached a commit point or has had its updates backed out since you last tried to retrieve the segment. If the retry fails, a GG status code is returned to your program.

For calls to Fast Path DEDBs, option T does not cause DL/I to retry the operation. A GG status code is returned. The T option must be specified as PROCOPT=GOT, GOT, or GOTP.

GOx and data integrity

For a very small set of applications and data, PROCOPT=GOx offers some performance and parallelism benefits. However, it does not offer application data integrity. For example, using PROCOPT=GOT in an online environment on a full-function database can cause performance degradation. The T option forces a re-read from DASD, negating the advantage of very large buffer pools and VSAM hiperspace for all currently running applications and shared data. For more information on the GOx processing option for DEDBs, see *IMS Version 11 System Utilities*.

Related concepts:

"Read without integrity"

Read without integrity

Database-level sharing of IMS databases provides for sharing of databases between a single update-capable batch or online IMS system and any number of other IMS systems that are reading data that are without integrity.

A GE status code might be returned to a program using PROCOPT=GOx for a segment that exists in a HIDAM database during control interval (CI) splits.

In IMS, programs that use database-level sharing include PROCOPT=GOx in their DBPCBs for that data. For batch jobs, the DBPCB PROCOPTs establish the batch job's access level for the database. That is, a batch job uses the highest declared

intent for a database as the access level for DBRC database authorization. In an online IMS environment, database ACCESS is specified on the DATABASE macro during IMS system definition, and it can be changed using the /START DB ACCESS=RO command. Online IMS systems schedule programs with data availability determined by the PROCOPTs within those program PSBs being scheduled. That data availability is therefore limited by the online system's database access.

The PROCOPT=GON and GOT options provide certain limited PCB status code retry for some recognizable pointer errors, within the data that is being read without integrity. In some cases, dependent segment updates, occurring asynchronously to the read-without-integrity IMS instance, do not interfere with the program that is reading that data without integrity. However, update activity to an average database does not always allow a read-without-integrity IMS system to recognize a data problem.

What read without integrity means

Each IMS batch or online instance has OSAM and VSAM buffer pools defined for it. Without locking to serialize concurrent updates that are occurring in another IMS instance, a read without integrity from a database data set fetches a copy of a block or CI into the buffer pool in storage. Blocks or CIs in the buffer pool can remain there a long time. Subsequent read without integrity of other blocks or CIs can then fetch more recent data. Data hierarchies and other data relationships between these different blocks or CIs can be inconsistent.

For example, consider an index database (VSAM KSDS), which has an index component and a data component. The index component contains only hierarchic control information, relating to the data component CI where a given keyed record is located. Think of this as the way that the index component CI maintains the high key in each data component CI. Inserting a keyed record into a KSDS data component CI that is already full causes a CI split. That is, some portion of the records in the existing CI are moved to a new CI, and the index component is adjusted to point to the new CI.

For example, suppose the index CI shows the high key in the first data CI as KEY100, and a split occurs. The split moves keys KEY051 through KEY100 to a new CI; the index CI now shows the high key in the first data CI as KEY050, and another entry shows the high key in the new CI as KEY100.

A program that is reading is without integrity, which already read the “old” index component CI into its buffer pool (high key KEY100), does not point to the newly created data CI and does not attempt to access it. More specifically, keyed records that exist in a KSDS at the time a read-without-integrity program starts might never be seen. In this example, KEY051 through KEY100 are no longer in the first data CI even though the “old” copy of the index CI in the buffer pool still indicates that any existing keys up to KEY100 are in the first data CI.

Hypothetical cases also exist where the deletion of a dependent segment and the insertion of that same segment type under a different root, placed in the same physical location as the deleted segment, can cause simple Get Next processing to give the appearance of only one root in the database. For example, accessing the segments under the first root in the database down to a level-06 segment (which had been deleted from the first root and is now logically under the last root) would then reflect data from the other root. The next and subsequent Get Next calls retrieve segments from the other root.

Read-only (PROCOPT=GO) processing does not provide data integrity.

Data set extensions

IMS instances with database-level sharing can open a database for read without integrity.

After the database is opened, another program that is updating that database can make changes to the data. These changes might result in logical and physical extensions to the database data set. Because the read-without-integrity program is not aware of these extensions, problems with the RBA (beyond end-of-data) can occur.

Related concepts:

"Preventing a program from updating data: processing options" on page 93

Chapter 6. Gathering requirements for message processing options

One of the tasks of application design is providing information about your application's requirements to the people in charge of designing and administering your IMS system.

Restriction: This information applies to DB/DC and DCCTL environments only.

Related concepts:

"Programming with the IMS Java dependent region resource adapter" on page 642

Identifying online security requirements

Security in an online system means protecting the data from unauthorized use through terminals. It also means preventing unauthorized use of both the IMS system and the application programs that access the database. For example, you do not want a program that processes paychecks to be available to everyone who can access the system.

The security mechanisms that IMS provides are signon, terminal, and password security.

Related reading: For an explanation of how to establish these types of security, see *IMS Version 11 System Administration*.

Limiting access to specific individuals: signon security

Signon security is available through Resource Access Control Facility (RACF®) or a user-written security exit routine. With signon security, individuals who want to use IMS must be defined to RACF or its equivalent before they are allowed access.

When a person signs on to IMS, RACF or security exits verify that the person is authorized to use IMS before access to IMS-controlled resources is allowed. This signon security is provided by the `/SIGN ON` command. You can also limit the transaction codes and commands that individuals are allowed to enter. You do this by associating an individual's user identification (USERID) with the transaction codes and commands.

LU 6.2 transactions contain the USERID.

Related reading: For more information on security, see *IMS Version 11 Communications and Connections*.

Limiting access for specific terminals: terminal security

Use terminal security to limit the entry of a transaction code to a particular terminal or group of terminals in the system. How you do this depends on how many programs you want to protect.

To protect a particular program, you can either authorize a transaction code to be entered from a list of logical terminals, or you can associate each logical terminal

© Copyright IBM Corp. 1974, 2015

with a list of the transaction codes that a user can enter from that logical terminal. For example, you could protect the paycheck application program by defining the transaction code associated with it as valid only when entered from the terminals in the payroll department. If you wanted to restrict access to this application even more, you could associate the paycheck transaction code with only one logical terminal. To enter that transaction code, a user needs to be at a physical terminal that is associated with that logical terminal.

Restriction: If you are using the shared-queues option, static control blocks representing the resources needed for the security check need to be available in the IMS system where the security check is being made. Otherwise, the security check is bypassed.

Related reading: For more information on shared queues, see *IMS Version 11 System Administration*.

Limiting access to the program: password security

Another way you can protect the application program is to require a password when a person enters the transaction code that is associated with the application program you want to protect. If you use only password security, the person entering a particular transaction code must also enter the password of the transaction before IMS processes the transaction.

If you use password security with terminal security, you can restrict access to the program even more. In the paycheck example, using password security and terminal security means that you can restrict unauthorized individuals within the payroll department from executing the program.

Restriction: Password security for transactions is only supported if the transactions that are needed for the security check are defined in the IMS system where the security check is being made. Otherwise, the security check is bypassed.

Allowing access to security data: authorization security

RACF has a data set that you can use to store user-unique information. The AUTH call gives application programs access to the RACF data set security data, and a way to control access to application-defined resources. Thus, application programs can obtain the security information about a particular user.

How IMS security relates to DB2 for z/OS security

An important part of DB2 for z/OS security is the authorization ID. The authorization ID that IMS uses for a program or a user at a terminal depends on the kind of security that is used and the kind of program that is running.

For MPPs, IFPs, and transaction-oriented BMPs, the authorization ID depends on the type of IMS security:

- If signon is required, IMS passes the USERID and group name that are signed-on to DB2 for z/OS.
- If signon is not required, DB2 for z/OS uses the name of the originating logical terminal as the authorization ID.

For batch-oriented BMPs, the authorization ID is dependent on the value specified for the BMPUSID= keyword in the DFSDCxxx PROCLIB member:

- If BMPUSID=USERID is specified, the value from the USER= keyword on the JOB statement is used.
- If USER= is not specified on the JOB statement, the program's PSB name is used.
- If BMPUSID=PSBNAME is specified, or if BMPUSID= is not specified at all, the program's PSB name is used.

Supplying security information

When you evaluate your application in terms of its security requirements, you need to look at each program individually. When you have done this, you can supply the following information to your security personnel.

- For programs that require signon security:
  - List the individuals who should be able to access IMS.
- For programs that require terminal security:
  - List the transaction codes that must be secured.
  - List the terminals that should be allowed to enter each of these transaction codes. If the terminals you are listing are already installed and being used, identify the terminals by their logical terminal names. If not, identify them by the department that will use them (for example, the accounting department).
- For programs that require password security:
  - List the transaction codes that require passwords.
- For commands that require security:
  - List the commands that require signon or password security.

Related concepts:

"An overview of application design" on page 15

Analyzing screen and message formats

When an application program communicates with a terminal, an editing procedure translates messages from the way they are entered at the terminal to the way the program expects to receive and process them.

The decisions about how IMS will edit your program's messages are based on how your data should be presented to the person at the terminal and to the application program. You need to describe how you want data from the program to appear on the terminal screen, and how you want data from the terminal to appear in the application program's I/O area. (The I/O area contains the segments being processed by the application program.)

To supply information that will be helpful in these decisions, you should be familiar with how IMS edits messages. IMS has two editing procedures:

- **Message Format Service (MFS)** uses control blocks that define what a message should look like to the person at the terminal and to the application program.
- **Basic edit** is available to all IMS application programs. Basic edit removes control characters from input messages and inserts the control characters you specify in output messages to the terminal.

Related reading: For information on defining IMS editing procedures and on other design considerations for IMS networks, see *IMS Version 11 Communications and Connections*.

An overview of MFS

MFS uses four kinds of control blocks to format messages between an application program and a terminal. The information you gather about how you want the data formatted when it is passed between the application program and the terminal is contained in these control blocks.

The two control blocks that describe input messages to IMS are:

- The device input format (DIF) describes to IMS what the input message is to look like when it is entered at the terminal.
- The message input descriptor (MID) tells IMS how the application program expects to receive the input message in its I/O area.

By using the DIF and the MID, IMS can translate the input message from the way that it is entered at the terminal to the way it should appear in the program's I/O area.

The two control blocks that describe output messages to IMS are:

- The message output descriptor (MOD) tells IMS what the output message is to look like in the program's I/O area.
- The device output format (DOF) tells IMS how the message should appear on the terminal.

To define the MFS control blocks for an application program, you need to know how you want the data to appear at the terminal and in the application program's I/O area for both input and output.

An overview of basic edit

Basic edit removes the control characters from an input message before the application program receives it, and inserts the control characters you specify when the application program sends a message back to the terminal.

To format output messages at a terminal using basic edit, you need to supply the necessary control characters for the terminal you are using.

If your application will use basic edit, you should describe how you want the data to be presented at the terminal, and what it is to look like in the program's I/O area.

Editing considerations in your application

Before you describe the editing requirements of your application, be sure that you are aware of your standards concerning screen design. Make sure that the requirements that you describe comply with those standards.

Provide the following information about your program's editing requirements:

- How you want the screen to be presented to the person at the terminal for the person to enter the input data. For example, if an airline agent wants to reserve seats on a particular flight, the screen that asks for this information might look like this:

  FLIGHT#:
  NAME:
  NO. IN PARTY:

- What the data should look like when the person at the terminal enters the input message.
- What the input message should look like in the program's I/O area.
- What the data should look like when the program builds the output message in its I/O area.
- How the output message should be formatted at the terminal.
- The length and type of data that your program and the terminal will be exchanging.

The type of data you are processing is only one consideration when you analyze how you want the data presented at the terminal. In addition, you should weigh the needs of the person at the terminal (the human factors aspects in your application) against the effect of the screen design on the efficiency of the application program (the performance factors in the application program). Unfortunately, sometimes a trade-off between human factors and performance factors exists. A screen design that is easily understood and used by the person at the terminal may not be the design that gives the application program its best performance. Your first concern should be that you are following whatever are your established screen standards.

A terminal screen that has been designed with human factors in mind is one that puts the person at the terminal first; it is one that makes it as easy as possible for that person to interact with IMS. Some of the things you can do to make it easy for the person at the terminal to understand and respond to your application program are:

- Display a small amount of data at one time.
- Use a format that is clear and uncluttered.
- Provide clear and simple instructions.
- Display one idea at a time.
- Require short responses from the person at the terminal.
- Provide some means for help and ease of correction for the person at the terminal.

At the same time, you do not want the way in which a screen is designed to have a negative effect on the application program's response time, or on the system's performance. When you design a screen with performance first in mind, you want to reduce the processing that IMS must do with each message. To do this, the person at the terminal should be able to send a lot of data to the application program in one screen so that IMS does not have to process additional messages. And the program should not require two screens to give the person at the terminal information that it could give on one screen.

When describing how the program should receive the data from the terminal, you need to consider the program logic and the type of data you are working with.

Gathering requirements for conversational processing

When you use *conversational processing*, the person at the terminal enters some information, and an application program processes the information and responds to the terminal. The person at the terminal then enters more information for an application program to process. Each of these interactions between the person at

the terminal and the program is called a *step* in the conversation. Only MPPs can be conversational programs; Fast Path programs and BMPs cannot be conversational.

Definition: Conversational processing means that the person at the terminal can communicate with the application program.

What happens in a conversation

A *conversation* is defined as a dialog between a user at a terminal and IMS through a scratchpad area (SPA) and one or more application programs.

During a conversation, the user at the terminal enters a request, receives the information from IMS, and enters another request. Although it is not apparent to the user, a conversation can be processed by several application programs or by one application program.

To continue a conversation, the program must have the necessary information to continue processing. IMS stores data from one step of the conversation to the next in a SPA. When the same program or a different program continues the conversation, IMS gives the program the SPA for the conversation associated with that terminal.

In the preceding airline example, the first program might save the flight number and the names of the people traveling, and then pass control to another application program to reserve seats for those people on that flight. The first program saves this information in the SPA. If the second application program did not have the flight number and names of the people traveling, it would not be able to do its processing.

Designing a conversation

The first part of designing a conversation is to design the flow of the conversation. If the requests from the person at the terminal are to be processed by only one application program, you need only to design that program. If the conversation should be processed by several application programs, you need to decide which steps of the conversation each program is to process, and what each program is to do when it has finished processing its step of the conversation.

When a person at a terminal enters a transaction code that has been defined as conversational, IMS schedules the conversational program (for example, Program A) associated with that transaction code. When Program A issues its first call to the message queue, IMS returns the SPA that is defined for that transaction code to Program A's I/O area. The person at the terminal must enter the transaction code (and password, if one exists) only on the first input screen; the transaction code need not be entered during each step of the conversation. IMS treats data in subsequent screens as a continuation of the conversation started on the first screen.

After the program has retrieved the SPA, Program A can retrieve the input message from the terminal. After it has processed the message, Program A can either continue the conversation, or end it.

To continue the conversation, Program A can do any of the following:

- Reply to the terminal that sent the message.
- Reply to the terminal and pass the conversation to another conversational program, for example Program B. This is called a *deferred program switch*.

Definition: A deferred program switch means that Program A responds to the terminal and then passes control to another conversational program, Program B. After passing control to Program B, Program A is no longer part of the conversation. The next input message that the person at the terminal enters goes to Program B, although the person at the terminal is unaware that this message is being sent to a second program.

Restriction: A deferred program switch is disallowed if the application is involved in an inbound protected conversation. The application will receive an X6 status code if it attempts to perform a deferred program switch in this environment.

- Pass control of the conversation to another conversational program without first responding to the originating terminal. This is called an *immediate program switch*.

  Definition: An immediate program switch lets you pass control directly to another conversational program without having to respond to the originating terminal. When you do this, the program that you pass the conversation to must respond to the person at the terminal. To continue the conversation, Program B then has the same choices as Program A did: It can respond to the originating terminal and keep control, or it can pass control in a deferred or immediate program switch.

  Restriction: An immediate program switch is disallowed if the application is involved in an inbound protected conversation. The application will be abended with a U711 if it attempts to perform an immediate program switch in this environment.

To end the conversation, Program A can do either of the following:

- Move a blank to the first byte of the transaction code area of the SPA and then return the SPA to IMS.
- Respond to the terminal and pass control to a nonconversational program. This is also called a deferred program switch, but Program A ends the conversation before passing control to another application program. The second application program can be an MPP or a transaction-oriented BMP that processes transactions from the conversational program.

Important points about the scratchpad area (SPA)

When program A passes control of a conversation to program B, program B needs to have the data that program A saved in the SPA in order to continue the conversation. IMS gives the SPA for the transaction to program B when program B issues its first message call.

The SPA is kept with the message. When the truncated data option is on, the size of the retained SPA is the largest SPA of any transaction in the conversation.

For example, if the conversation starts with TRANA (SPA=100), and the program switches to a TRANB (SPA=50), the input message for TRANB will contain a SPA segment of 100 bytes. IMS adjusts the size of the SPA so that TRANB receives only the first 50 bytes.

Recovery considerations in conversations

Because a conversation involves several steps and can involve several application programs, consider the following items.

- One way you can make recovery easier is to design the conversation so that all the database updates are done in the last step of the conversation. This way, if

the conversation terminates abnormally, IMS can back out all the updates because they were all made during the same step of the conversation. Updating the database during the last step of the conversation is also a good idea, because the input from each step of the conversation is available.

- Although a conversation can terminate abnormally during any step of the conversation, IMS backs out only the database updates and output messages resulting during the last step of the conversation. IMS does not back out database updates or cancel output messages for previous steps, even though some of that processing might be inaccurate as a result of the abnormal termination.
- Certain IMS system service calls can be helpful if the program determines that some of its processing was invalid. These calls include ROLB, SETS, SETU, and ROLS. The Roll Back call (ROLB) backs out all of the changes that the program has made to the database. ROLB also cancels the output messages that the program has created (except those sent with an express PCB) since the program's last commit point.

  The SETS, or SETU, and ROLS (with a *token*) calls work together to allow the application program to set intermediate backout points within the call processing of the program. The application program can set up to nine intermediate backout points. Your program needs to use the SETS or SETU call to specify a token for each point. A subsequent ROLS call, using the same token, can back out all database changes and discard all nonexpress messages processed since that SETS or SETU call.

  Definition: A token is a 4-byte identifier.
- The program can use an express PCB to send a message to the person at the terminal and to the master terminal operator. When the application program inserts messages using an express PCB, IMS waits until it has the complete message, rather than for the occurrence of a commit point, to transmit the message to its destination. (In this context, "insert" refers to a situation in which the application program sends the message and it is received by IMS; "transmit" refers to a situation in which IMS begins sending the message to its destination.) Therefore, when IMS has the complete message, it will be transmitted even if the program abnormally terminates. Messages sent with an express PCB are sent to their final destinations even if the program terminates abnormally or issues a ROLB call.
- To verify the accuracy of the previous processing, and to correct the processing that is determined to be inaccurate, you can use the Conversational Abnormal termination routine, DFSCONE0.

  Related reading: For more information on DFSCONE0, see *IMS Version 11 Exit Routines*.
- You can write an MPP to examine the SPA, send a message notifying the person at the terminal of the abnormal termination, make any necessary database calls, and use a user-written or system-provided exit routine to schedule it.

Related concepts:

"To other programs and terminals" on page 107

Identifying output message destinations

An application program can send messages to another application program or to IMS terminals. To send output messages, the program issues a call and references the I/O PCB or an alternate PCB. The I/O PCB and alternate PCBs represent logical terminals and other application programs with which the application program communicates.

Definition: An *alternate PCB* is a data communication program communication block (DCPCB) that you define to describe output message destinations other than the terminal that originated the input message.

Related concepts:

"Batch message processing: transaction-oriented BMPs" on page 44

The originating terminal

To send a message to the logical terminal that sent the input message, the program uses an I/O PCB. IMS puts the name of the logical terminal that sent the message in the I/O PCB when the program receives the message.

As a result, the program need not do anything to the I/O PCB before sending the message. If a program receives a message from a batch-oriented BMP or CPI Communications driven program, no logical terminal name is available to put into the I/O PCB. In these cases, the logical terminal name field contains blanks.

Related concepts:

"Identifying output message destinations" on page 106

To other programs and terminals

When you want to send an output message to a terminal other than, or in addition to, the terminal that sent the input message, you use an alternate PCB. You can set the alternate PCB for a specific logical terminal when the program's PSB is generated, or you can define the alternate PCB as being modifiable. A program can change the destination of a modifiable alternate PCB while the program is running, so you can send output messages to several alternate destinations.

The application program might need to respond to the originating terminal before the person at the originating terminal can send any more messages. This might occur when a terminal is in **response mode** or in **conversational mode**:

- **Response mode** can apply to a communication line, a terminal, or a transaction. When response mode is in effect, IMS does not accept any input from the communication line or terminal until the program has sent a response to the previous input message. The originating terminal is unusable (for example, the keyboard locks) until the program has processed the transaction and sent the reply back to the terminal.

  If a response-mode transaction is processed, including Fast Path transactions, and the application does not insert a response back to the terminal through either the I/O PCB or alternate I/O PCB, but inserts a message to an alternate PCB (program-to-program switch), the second or subsequent application program must respond to the originating terminal and satisfy the response. IMS will not take the terminal out of response mode.

  If an application program terminates normally and does not issue an `ISRT` call to the I/O PCB, alternate I/O PCB, or alternate PCB, IMS sends system message DFS2082I to the originating terminal to satisfy the response for all response-mode transactions, including Fast Path transactions.

  You can define communication lines and terminals as operating in response mode, not operating in response mode, or operating in response mode only if processing a transaction that is been defined as response mode. You specify response mode for communication lines and terminals on the TYPE and TERMINAL macros, respectively, at IMS system definition. You can define any transaction as a response-mode transaction; you do this on the TRANSACT macro at IMS system definition. Response mode is in effect if:

  – The communication line has been defined as being in response mode.

  - The terminal has been defined as being in response mode.
  - The transaction code has been defined as response mode.
- **Conversational mode** applies to a transaction. When a program is processing a conversational transaction, the program must respond to the originating terminal after each input message it receives from the terminal.

In these processing modes, the program must respond to the originating terminal. But sometimes the originating terminal is a physical terminal that is made up of two components—for example, a printer and a display. If the physical terminal is made up of two components, each component has a different logical terminal name. To send an output message to the printer part of the terminal, the program must use a different logical terminal name than the one associated with the input message; it must send the output message to an alternate destination. A special kind of alternate PCB is available to programs in these situations; it is called an *alternate response PCB*.

Definition: An alternate response PCB lets you send messages when exclusive, response, or conversational mode is in effect. See the next section for more information.

Alternate response PCB

The destination of an alternate response PCB must be a logical terminal—you cannot use an alternate response PCB to represent another application program. When you use an alternate response PCB during response mode or conversational mode, the logical terminal represented by the alternate response PCB must represent the same physical terminal as the originating logical terminal.

In these processing modes, after receiving the message, the application program must respond by issuing an `ISRT` call to one of the following:

- The I/O PCB.
- An alternate response PCB.
- An alternate PCB whose destination is another application program, that is, a program-to-program switch.
- An alternate PCB whose destination is an ISC link. This is allowed only for front-end switch messages.

  Related reading: For more information on front-end switch messages, see *IMS Version 11 Exit Routines*.

If one of these criteria is not met, message DFS2082I is sent to the terminal.

Express PCB

Consider specifying an alternate PCB as an *express PCB*. The express designation relates to whether a message that the application program inserted is actually transmitted to the destination if the program abnormally terminates or issues a `ROLL`, `ROLB`, or `ROLS` call. For all PCBs, when a program abnormally terminates or issues a `ROLL`, `ROLB`, or `ROLS` call, messages that were inserted but not made available for transmission are cancelled while messages that were made available for transmission are never cancelled.

Definition: An express PCB is an alternate response PCB that allows your program to transmit the message to the destination terminal earlier than when you use a nonexpress PCB.

For a nonexpress PCB, the message is not made available for transmission to its destination until the program reaches a commit point. The commit point occurs when the program terminates, issues a CHKP call, or requests the next input message and when the transaction has been defined with MODE=SNGL.

For an express PCB, when IMS has the complete message, it makes the message available for transmission to the destination. In addition to occurring at a commit point, it also occurs when the application program issues a PURG call using that PCB or when it requests the next input message.

You should provide the answers to the following questions to the data communications administrator to help in meeting your application's message processing requirements:

- Will the program be required to respond to the terminal before the terminal can enter another message?
- Will the program be responding only to the terminal that sends input messages?
- If the program needs to send messages to other terminals or programs as well, is there only one alternate destination?
- What are the other terminals to which the program must send output messages?
- Should the program be able to send an output message before it terminates abnormally?

Related concepts:

“Recovery considerations in conversations” on page 105

“Identifying output message destinations” on page 106

Chapter 7. Designing an application for APPC

Advanced Program-to-Program Communication (APPC) is IBM's preferred protocol for program-to-program communication. Application programs can be distributed throughout the network and communicate with each other in many hardware architectures and software environments.

Related Reading: For more information on APPC, see:

- *IMS Version 11 Communications and Connections*, which includes an overview of APPC for LU 6.2 devices and CPI Communications concepts.

Overview of APPC and LU 6.2

APPC allows application programs using APPC protocols to enter IMS transactions from LU 6.2 devices. The LU 6.2 application program runs on an LU 6.2 device supporting APPC.

APPC creates an environment that allows:

- Remote LU 6.2 devices to enter IMS local and remote transactions
- IMS application programs to insert transaction output to LU 6.2 devices with no coding changes to existing application programs
- New application programs to make full use of LU 6.2 device facilities
- Data integrity provided by IMS and in LU 6.2 environments that do not have a distributed sync-point function

Application program types

APPC/IMS is part of IMS TM that uses the CPI communications interface to communicate with application programs.

APPC/IMS supports the following types of application programs for LU 6.2 processing:

- Standard DL/I
- Modified standard DL/I
- CPI Communications driven

Standard DL/I application program

A standard DL/I application program does not issue any CPI Communications calls or establish any CPI-C conversations. This application program can communicate with LU 6.2 products that replace other LU-type terminals using the IMS API. A standard DL/I application program does not need to be modified, recompiled, or bound, and it executes as it currently does.

Modified standard DL/I application program

A modified standard DL/I application program is a standard DL/I online IMS TM application program that uses both DL/I calls and CPI Communications calls. It can be an MPP, BMP, or IFP that can access full-function databases, DEDBs, MSDBs, and DB2 for z/OS databases.

© Copyright IBM Corp. 1974, 2015

A modified standard DL/I application program uses CPI Communications (CPI-C) calls to provide support for an LU 6.2 and non-LU 6.2 mixed network. The same application program can be a standard DL/I on one execution, when the CPI Communications `ALLOCATE` verb is not issued, and a modified standard DL/I on a different execution when the CPI Communications `ALLOCATE` verb is issued.

A modified standard DL/I application program receives its messages using DL/I `GU` calls to the I/O PCB and issues output responses using DL/I `ISRT` calls. CPI Communications calls can also be used to allocate new conversations and to send and receive data for them.

Related Reading: For a list of the CPI Communications calls, see CPI Communications Reference.

Use a modified standard DL/I application program when you want to use an existing standard DL/I application program to establish a conversation with another LU 6.2 device or the same network destination. The standard DL/I application program is optionally modified and uses new functions, new application and transaction definitions, and modified DL/I calls to initiate LU 6.2 application programs. Program calls and parameters are available to use the IMS-provided implicit API and the CPI Communications explicit API.

CPI Communications driven program

A CPI Communications driven application program uses `Commit` and `Backout` calls, and CPI Communications interface calls or LU 6.2 verbs for input and output message processing. This application program uses the CPI Communications explicit API, and can access full-function databases, DEDBs, MSDBs, and DB2 for z/OS databases. An LU 6.2 device can activate a CPI Communications driven application program only by allocating a conversation.

Unlike a standard DL/I or modified standard DL/I application program, input and output message processing for a CPI Communications driven program uses APPC/MVS buffers and bypasses IMS message queueing. Because these application programs do not use the IMS message queue, they can control their own execution with the partner LU 6.2 system. An IMS `APSB` call enables you to allocate a PSB for accessing IMS databases and alternate PCBs.

The application program uses the Common Programming Interface Resource Recovery (CPI-RR) `SRRCMIT` verb to initiate an IMS sync point and the CPI-RR `SRRBACK` verb for backout. CPI Communications driven application programs use the CPI-RR calls to initiate IMS sync point processing prior to program termination.

A CPI Communications driven application program is able to:

- Access any type of database
- Receive and send large messages like the standard DL/I and modified standard DL/I application programs
- Control the flow of input and output with CPI Communications calls
- Allocate multiple conversations with partner LU 6.2 devices
- Cause synchronization with conversation partners
- Use the IMS implicit API (for example, IMS queue services)
- Use IMS services (for example, sync point at program termination) regardless of the API that is used

Application objectives

Each application type has a different purpose, and its ease-of-use varies depending on whether the program is a standard DL/I, modified standard DL/I, or a CPI Communications driven application program.

The following table lists the purpose and ease-of-use for each application type (standard DL/I, modified standard DL/I, and PI-C driven). This information must be balanced with IMS resource use.

Table 26. Using application programs in APPC.

	Ease of use		
Purpose of application program	**Standard DL/I program**	**Modified standard DL/I program**	**PI-C driven program**
Inquiry	Easy	Neutral	Very Difficult
Data Entry	Easy	Easy	Difficult
Bulk Transfer	Easy	Easy	Neutral
Cooperative	Difficult	Difficult	Desirable
Distributed	Difficult	Neutral	Desirable
High Integrity	Neutral	Neutral	Desirable
Client Server	Easy	Neutral	Very Difficult

Choosing conversation attributes

The LU 6.2 transaction program indicates how the transaction is to be processed by IMS. Two processing modes are available: **synchronous** and **asynchronous**.

Synchronous conversation

A conversation is synchronous if the partner waits for the response on the same conversation used to send the input data.

Synchronous processing is requested by issuing the `RECEIVE_AND_WAIT` verb after the `SEND_DATA` verb. Use this mode for IMS response-mode transactions and IMS conversational-mode transactions.

Example:

```
MC_ALLOCATE TPN(MYTXN)
MC_SEND_DATA  'THIS CAN BE A RESPONSE MODE'
MC_SEND_DATA  'OR CONVERSATIONAL MODE'
MC_SEND_DATA  'IMS TRANSACTION'
MC_RECEIVE_AND_WAIT
```

Asynchronous conversation

A conversation is asynchronous if the partner program normally deallocates a conversation after sending the input data. Output is sent to the TP name of DFSASYNC.

Asynchronous processing is requested by issuing the `DEALLOCATE` verb after the `SEND_DATA` verb. Use asynchronous processing for IMS commands, message switches, and non-response, non-conversational transactions.

Example:

```
MC_ALLOCATE TPN(OTHERTXN)
MC_SEND_DATA  'THIS MUST BE A MESSAGE SWITCH, IMS COMMAND'
MC_SEND_DATA  'OR A NON-RESP NON-CONV TRANSACTION'
MC_DEALLOCATE
```

Asynchronous output delivery

Asynchronous output is held on the IMS message queue for delivery. When the output is enqueued, IMS attempts to allocate a conversation to send this output. If this fails, IMS holds the output for later delivery. This delivery can be initiated by an operator command (/ALLOC), or by the enqueue of a new message for this LU 6.2 destination.

MSC synchronous and asynchronous conversation

MSC remote application messages from both synchronous and asynchronous APPC conversations can be queued on the multiple systems coupling (MSC) link. These messages can then be sent across the MSC link to a remote IMS for processing.

Related concepts:

"LU 6.2 flow diagrams" on page 122

Conversation type

The APPC conversation type defines how data is passed on and retrieved from APPC verbs.

It is similar in concept to file blocking and affects both ends of the conversation.

APPC supports two types of conversations:

Basic conversation

This low-conversation allows programs to exchange data in a standardized format. This format is a stream of data containing 2-byte length fields (referred to as LLs) that specify the amount of data to follow before the next length field. The typical data pattern is:

```
LL, data, LL, data
```

Each grouping of `LL, data` is referred to as a logical record. A basic conversation is used to send multiple segments with one verb and to receive maximum data with one verb.

Mapped conversation

This high-conversation allows programs to exchange arbitrary data records in data formats approved by application programmers. One send verb results in one receive verb, and z/OS and VTAM® handle the buffering.

Related Reading: For more information on basic and mapped conversations, see

- *Systems Network Architecture: LU 6.2 Reference: Peer Protocols* and
- *Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2*

Conversation state

CPI Communications uses conversation state to determine what the next set of actions will be.

Examples of conversation states are:

RESET
The initial state before communications begin.

SEND The program can send or optionally receive.

RECEIVE
The program must receive or abort.

CONFIRM
The program must respond to a partner.

The basic rules for APPC verbs are:

- The program that initiates the conversation speaks first.
- Only one APPC verb can be outstanding at time.
- Programs take turns sending and receiving.
- The state of the conversation determines the verbs a program can issue.

Synchronization level

The APPC synchronization level defines the protocol that is used when changing conversation states.

APPC and IMS support the following synchronization level values:

SYNCLVL=NONE
Specifies that the programs do not issue calls or recognize returned parameters relating to synchronization.

SYNCLVL=CONFIRM
Specifies that the programs can perform confirmation processing on the conversation.

SYNCLVL=SYNCPT
Specifies that the programs participate in coordinated commit processing on resources that are updated during the conversation under the z/OS Resource Recovery Services (RRS) recovery platform. A conversation with this level is also called a *protected conversation*.

Allocating a conversation with SYNCLVL=SYNCPT requires the RRS as the synchronization point manager. RRS controls the commitment of protected resources by coordinating the commit or backout request with the participating owners of the updated resources, the resource managers. IMS is the resource manager for DL/I, Fast Path data, and the IMS message queues. The application program decides whether the data is to be committed or aborted and communicates this decision to the synchronization point manager. The synchronization point manager then coordinates the actions in support of this decision among the resource managers.

Related concepts:

Activating protected conversations (Communications and Connections)

Introduction to resource recovery

Most customers maintain computer resources that are essential to the survival of their businesses. When these resources are updated in a controlled and synchronized manner, they are said to be **protected resources** or **recoverable resources**. These resources can all reside locally (on the same system) or be distributed (across nodes in the network). The protocols and mechanisms for regulating the updating of multiple protected resources in a consistent manner is provided in z/OS with z/OS Resource Recovery Services (RRS).

Participants in resource recovery

As shown in the following figure, the Resource Recovery environment is composed of three participants:

- Sync-point manager
- Resource managers
- Application program

RRS is the **sync-point manager**, also known as the coordinator. The sync-point manager controls the commitment of protected resources by coordinating the commit request (or backout request) with the **resource managers**, the participating owners of the updated resources. These resource managers are known as participants in the sync-point process. IMS participates as a resource manager for DL/I, Fast Path, and DB2 for z/OS data if this data has been updated in such an environment.

The final participant in this resource recovery protocol is the **application program**, the program accessing and updating protected resources. The application program decides whether the data is to be committed or aborted and relates this decision to the sync-point manager. The sync-point manager then coordinates the actions in support of this decision among the resource managers.

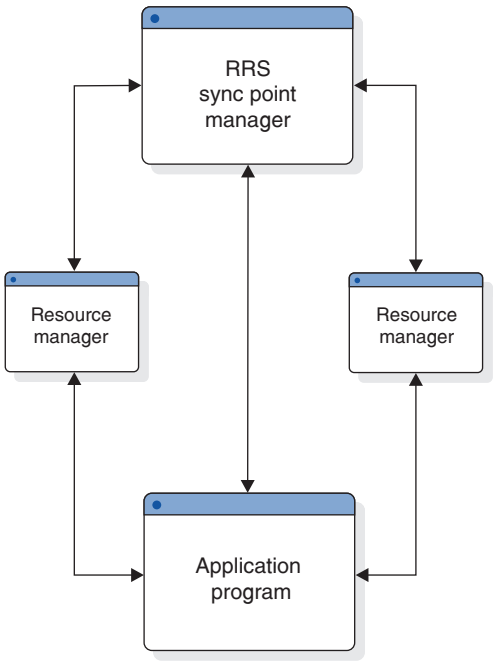


Figure 25. Participants in resource recovery

Two-phase commit protocol

As shown in the following figure, the two-phase commit protocol is a process involving the sync-point manager and the resource manager participants to ensure that of the updates made to a set of resources by a third participant, the application program, either **all updates occur or none**. In simple terms, the application program decides to commit its changes to some resources; this commit is made to the sync-point manager that then polls all of the resource managers as to the feasibility of the commit call. This is the prepare phase, often called phase one. Each resource manager votes yes or no to the commit.

After the sync-point manager has gathered all the votes, phase two begins. If all votes are to commit the changes, then the phase two action is commit. Otherwise, phase two becomes a backout. System failures, communication failures, resource manager failures, or application failures are not barriers to the completion of the two-phase commit process.

The work done by various resource managers is called a *unit of recovery (UOR)* and spans the time from one consistent point of the work to another consistent point, usually from one commit point to another. It is the unit of recovery that is the object of the two-phase commit process.

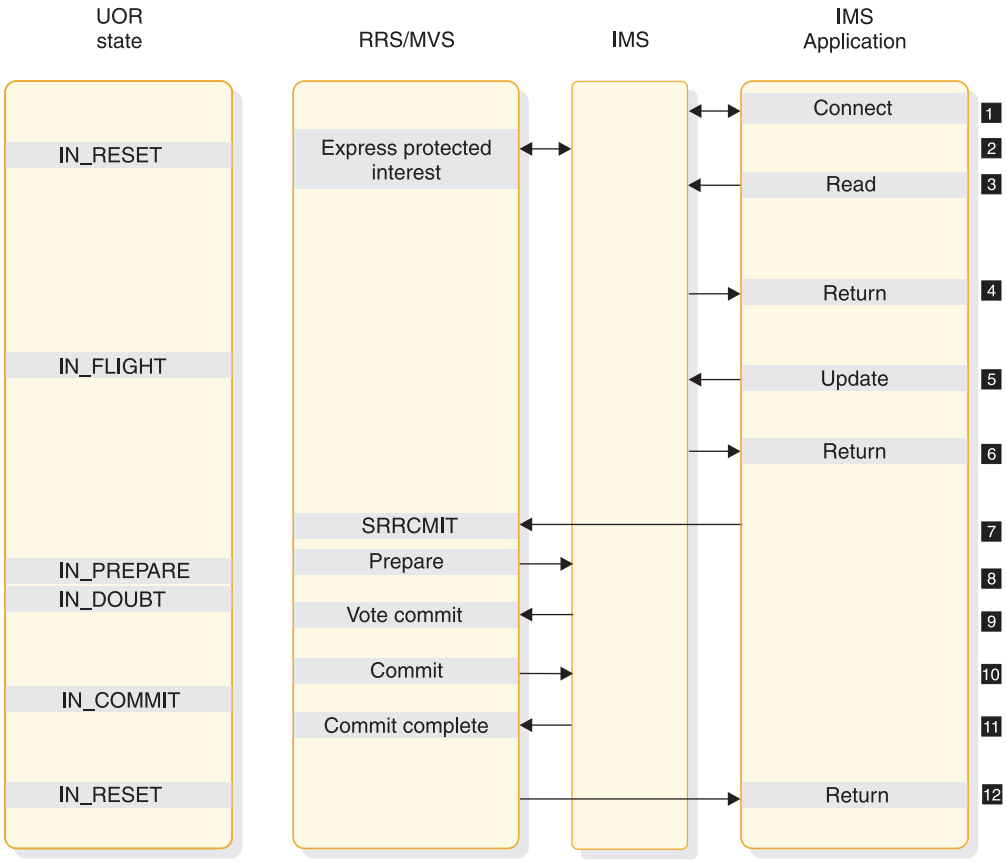


Figure 26. Two-phase commit process with one resource manager

Notes:

1. The application and IMS make a connection.
2. IMS expresses protected interest in the work started by the application. This tells RRS that IMS will participate in the 2-phase commit process.
3. The application makes a read request to an IMS resource.
4. Control is returned to the application following its read request.
5. The application updates a protected resource.
6. Control is returned to the application following its update request.
7. The application requests that the update be made permanent by way of the SRRCMIT call.
8. RRS calls IMS to do the prepare (phase 1) process.
9. IMS returns to RRS with its vote to commit.
10. RRS calls IMS to do the commit (phase 2) process.
11. IMS informs RRS that it has completed phase 2.
12. Control is returned to the application following its commit request.

Local versus distributed

The residence of the participants involved in the recovery process determines whether that recovery is considered local or distributed. In a **local** recovery scenario, all the participants reside on the same single system. In a **distributed** recovery scenario, the participants are scattered over multiple systems. The following figure shows the communication between Resource Manager participants

in a distributed resource recovery. There is no conceptual difference between a local and distributed recovery in the functions provided by RRS. However, to distribute the original sync-point manager's function to involve remote sync-point managers, a special resource manager is required. The APPC communications resource manager provides this support in the distributed environment.

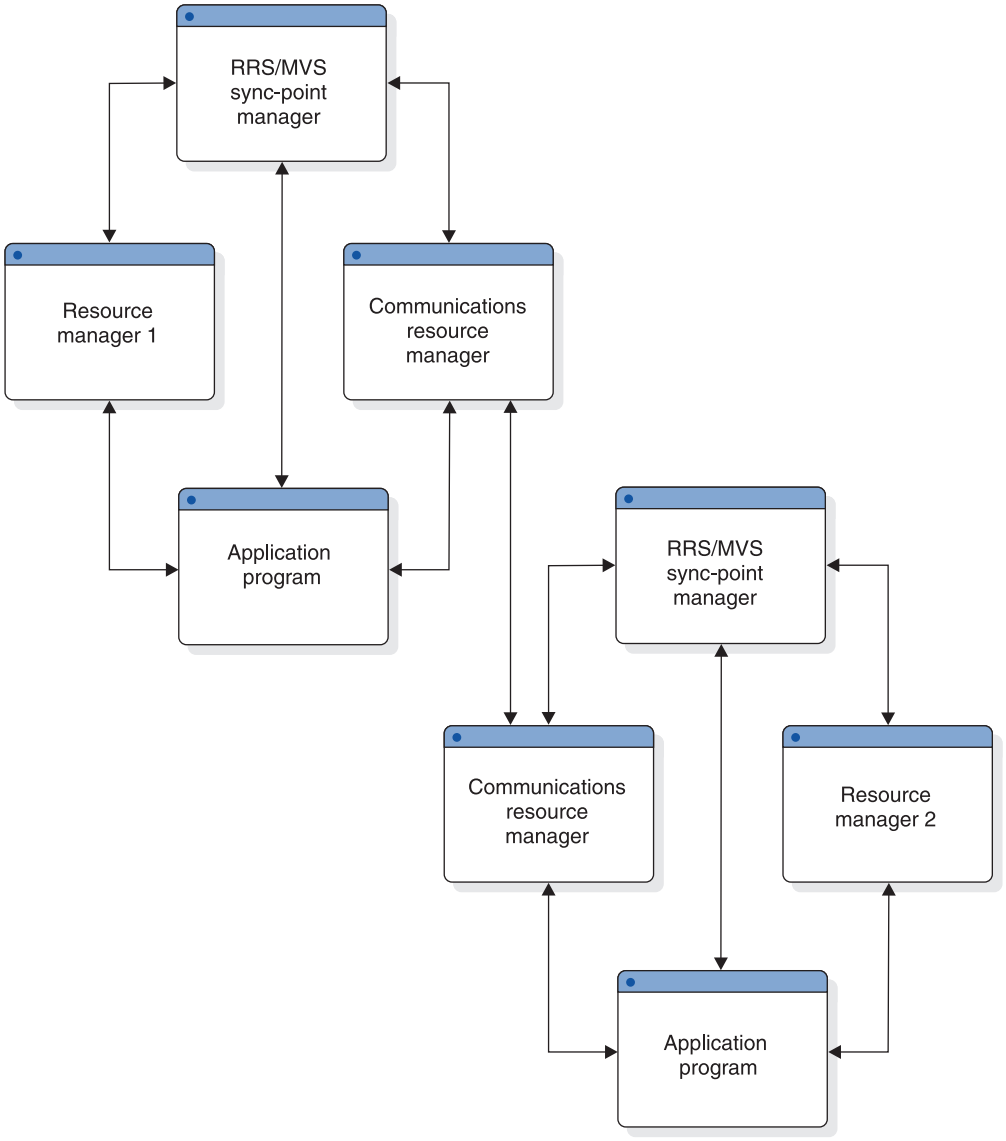


Figure 27. Distributed resource recovery

Summary of z/OS Resource Recovery Services support

z/OS Resource Recovery Services (RRS) provides a system resource recovery platform so that applications running on z/OS can access local and distributed resources and have system coordinated recovery management of these resources.

The support includes:

- A synchronization point manager to coordinate the two-phase commit process
- Implementation of the SAA Commit and Backout callable services for use by application programs

- A mechanism to associate resources with an application instance
- Services for resource manager registration and participation in the two-phase commit process with RRS
- Services to allow resource managers to express interest in an application instance and be informed of commit and backout requests
- Services to enable resource managers to obtain system data to restore their resources to consistent state
- A communications resource manager (called APPC/PC for APPC/Protected Conversations) so that distributed applications can coordinate their recovery with participating local resource managers

Restrictions:

- Extended Recovery Facility (XRF)

  Running protected conversations in an IMS-XRF environment does not guarantee that the alternate system can resume and resolve any unfinished work started by the active system. This process is not guaranteed because a failed resource manager must re-register with its original RRS system if the RRS is still available when the resource manager restarts. Only if the RRS on the active system is not available can an XRF alternate can register with another RRS in the sysplex and obtain the incomplete unit of recovery data of the failing active.

  Recommendation: Because IMS retains indoubt units-of-recovery indefinitely until they are resolved, switch back to the original active system as soon as possible to pick up unit-of -recovery information to resolve and complete all the work of the resource managers involved. If this is not possible, the indoubt units-of-recovery can be resolved using commands.
- Remote Site Recovery (RSR)

  Active systems tracked by a remote system in an RSR environment can participate in protected conversations, although it will be necessary to resolve indoubt units-of-recovery using commands if they exist after a takeover to a remote site has been done. This is because the remote site is probably not part of the active sysplex and the new IMS cannot acquire unfinished unit-of-recovery information from RRS. IMS provides commands to interrogate protected conversation work and to resolve the unfinished unit-of-recovery, if necessary.
- Batch and non-message-driven BMPs in a DBCTL Environment

  Distributed Sync Point does not support the IMS batch environment. In a DBCTL environment, inbound protected conversations are not possible. However, a BMP in a DBCTL environment can allocate an outbound protected conversation, which will be supported by Distributed Sync Point and RRS.

Distributed sync point

The Distributed Sync Point support enables IMS and remote application programs (APPC or OTMA) to participate in protected conversations with coordinated resource updates and recoveries. Before this support, IMS acted as the sync-point manager. In this new scenario, z/OS manages the sync-point process on behalf of the conversation participants: the application program and IMS (now acting as a resource manager).

z/OS implements a system resource recovery platform, the z/OS Resource Recovery Services (RRS). RRS supports the Common Programming Interface - Resource Recovery (CPI-RR), an element of the SAA Common Programming Interface that defines resource recovery and provides for the coordinated

management of resource recovery for both local *and* distributed resources. In addition to RRS, a communications resource manager (called APPC/PC for APPC/Protected Conversations) provides distribution of the recovery.

In the APPC environment, a protected conversation is initiated when the application program allocates an APPC conversation with `SYNC_LEVEL=SYNCPT`. Both IMS and APPC are resource managers in this scenario. In the OTMA environment, some additional code is required because OTMA is not a resource manager. The additional code needed is an OTMA adapter, IBM supplied or equivalent. This adapter indicates to IMS (in the OTMA message prefix) that this message is part of a protected conversation, and thus IMS and the adapter are participants in the coordinated commit process as managed by RRS.

Application programmers can now develop APPC application programs (local and remote) and remote OTMA application programs that use RRS as the sync-point manager, rather than IMS. This enhancement enables resources across multiple platforms to be updated and recovered in a coordinated manner.

Distributed sync point concepts

The Distributed Sync Point support entails:

- Changes in IMS that allow it to function as a resource manager under RRS
- Changes to the application program environment that support using applications in protected conversations
- Changes to some commands that aid the user

Impact on the network

Network traffic will increase as a result of the conversation participants and the sync-point manager communicating with each other.

Application programming interface for LU type 6.2

IMS application programs can use the IMS implicit LU 6.2 API to access LU 6.2 devices. This API provides compatibility with non-LU 6.2 device types so that the same application program can be used from both LU 6.2 and non-LU 6.2 devices.

The API adds to the APPC interface by supplying IMS-provided processing for the application program. You can use the explicit CPI Communications interface for APPC functions and facilities for new or rewritten IMS application programs.

Implicit API

The implicit API accesses an APPC conversation indirectly. This API uses the standard DL/I calls (`GU`, `ISRT`, `PURG`) to send and receive data. It allows application programs that are not specific to LU 6.2 protocols to use LU 6.2 devices.

The API uses new and changed DL/I calls (`CHNG`, `INQY`, `SETO`) to utilize LU 6.2. Using the existing IMS application programming base, you can write specific applications for LU 6.2 using this API and not using the CPI Communications calls. Although the implicit API uses only some of the LU 6.2 capabilities, it can be a useful simplification for many applications. The implicit API also provides function outside of LU 6.2, like message queueing and automatic asynchronous message delivery.

IMS generates all CPI Communications calls under the implicit API. The application interaction is strictly with the IMS message queue.

The remote LU 6.2 system must be able to handle the LU 6.2 flows. APPC/MVS generates these flows from the CPI Communications calls issued by the IMS application program using the implicit API. An IMS application program can use the explicit API to issue the CPI Communications directly. This is useful with remote LU 6.2 systems that have incomplete LU 6.2 implementations, or that are incompatible with the IMS implicit API support.

The existing API is extended so that:

- Asynchronous LU 6.2 output is created by using alternate PCBs that reference LU 6.2 destinations. The DL/I `CHNG` call can supply parameters to specify an LU 6.2 destination. Default values are used for omitted parameters.
- An application program can retrieve the current conversation attributes such as the conversation type (basic or mapped), the sync_level (NONE, CONFIRM, or SYNCPT), and asynchronous or synchronous conversation.
- A terminal message switch can be used to and from LU 6.2 devices.

Explicit API

The explicit API (the CPI Communications API) can be used by any IMS application program to access an APPC conversation directly.

IMS resources are available to the CPI Communications driven application program only if the application issues the `APSB` (Allocate PSB) call. The CPI Communications driven application program must use the CPI-RR `SRRCMIT` and `SRRBACK` verbs to initiate an IMS sync point or backout, or if SYNCLVL=SYNCPT is specified, to communicate the sync point decision to the z/OS Resource Recovery Services sync point manager.

Related Reading: For a description of the `SRRCMIT` and `SRRBACK` verbs, see *SAA CPI Resource Recovery Reference*.

LU 6.2 partner program design

The flow of a transaction that is sent from an LU 6.2 device differs, depending on the conversation attributes and synchronization levels. Different results occur, and the partner system takes actions accordingly.

LU 6.2 flow diagrams

The following diagrams show the flows for transactions that are sent from an LU 6.2 device.

The following figures show:

- The flow between a synchronous or asynchronous LU 6.2 application program and an IMS application program in a single (local) IMS system
- The flow between a synchronous or asynchronous LU 6.2 application program in a single (local) IMS system and an IMS application program in a remote IMS system across a multiple systems coupling (MSC) link
- A backout scenario with SYNC_LEVEL=SYNCPT

Differences in buffering and encapsulation of control data with user data may cause variations in the flows. The control data are the 3 returned fields from the

Receive APPC verb: Status_received, Data_received, and Request_to_send_received. Any variations based on these differences will not affect the function or use of the flows.

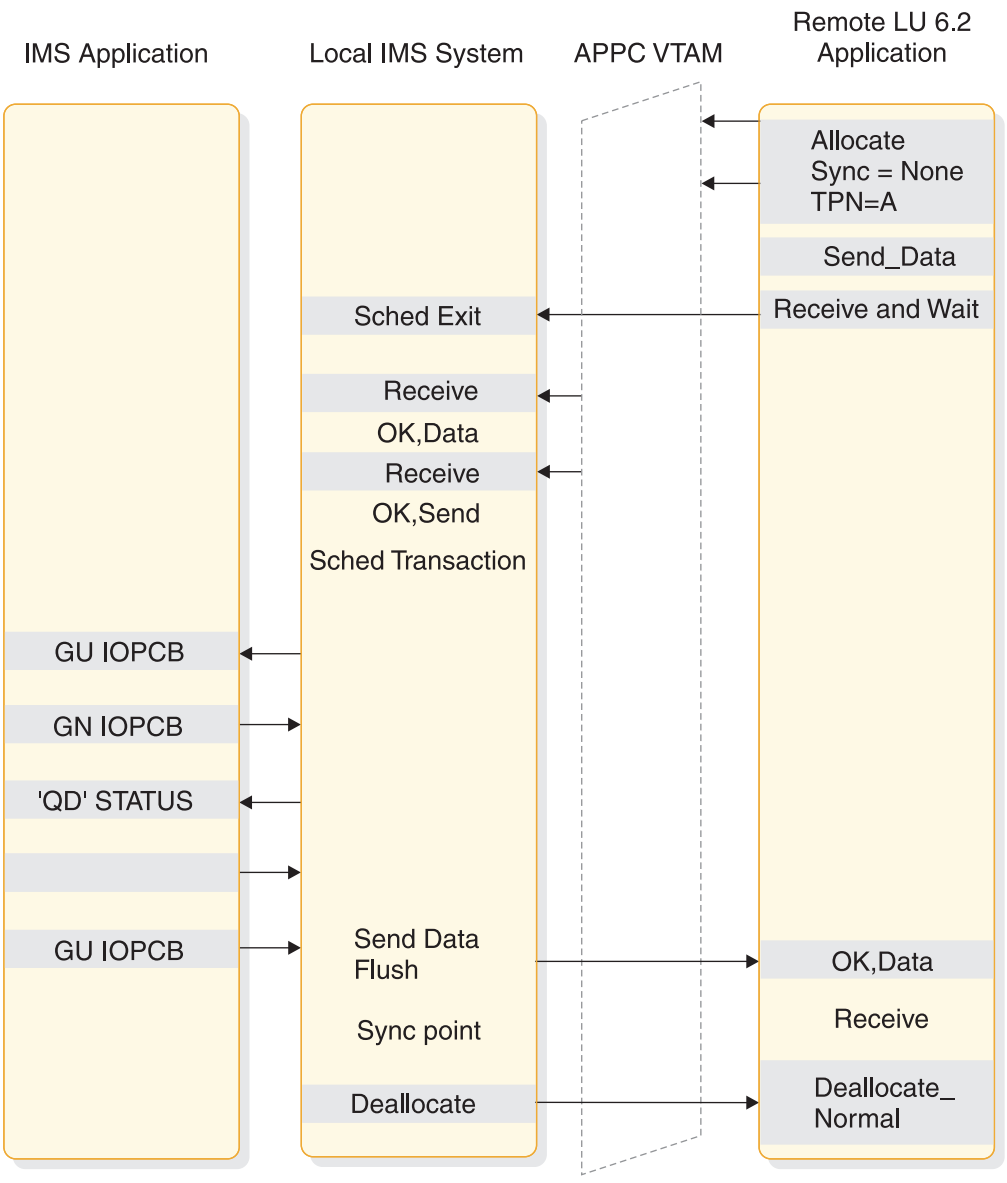


Figure 28. Flow of a local IMS synchronous transaction when Sync_level=None

Figure 29 on page 124 shows the flow of a local synchronous transaction when Sync_level is Confirm.

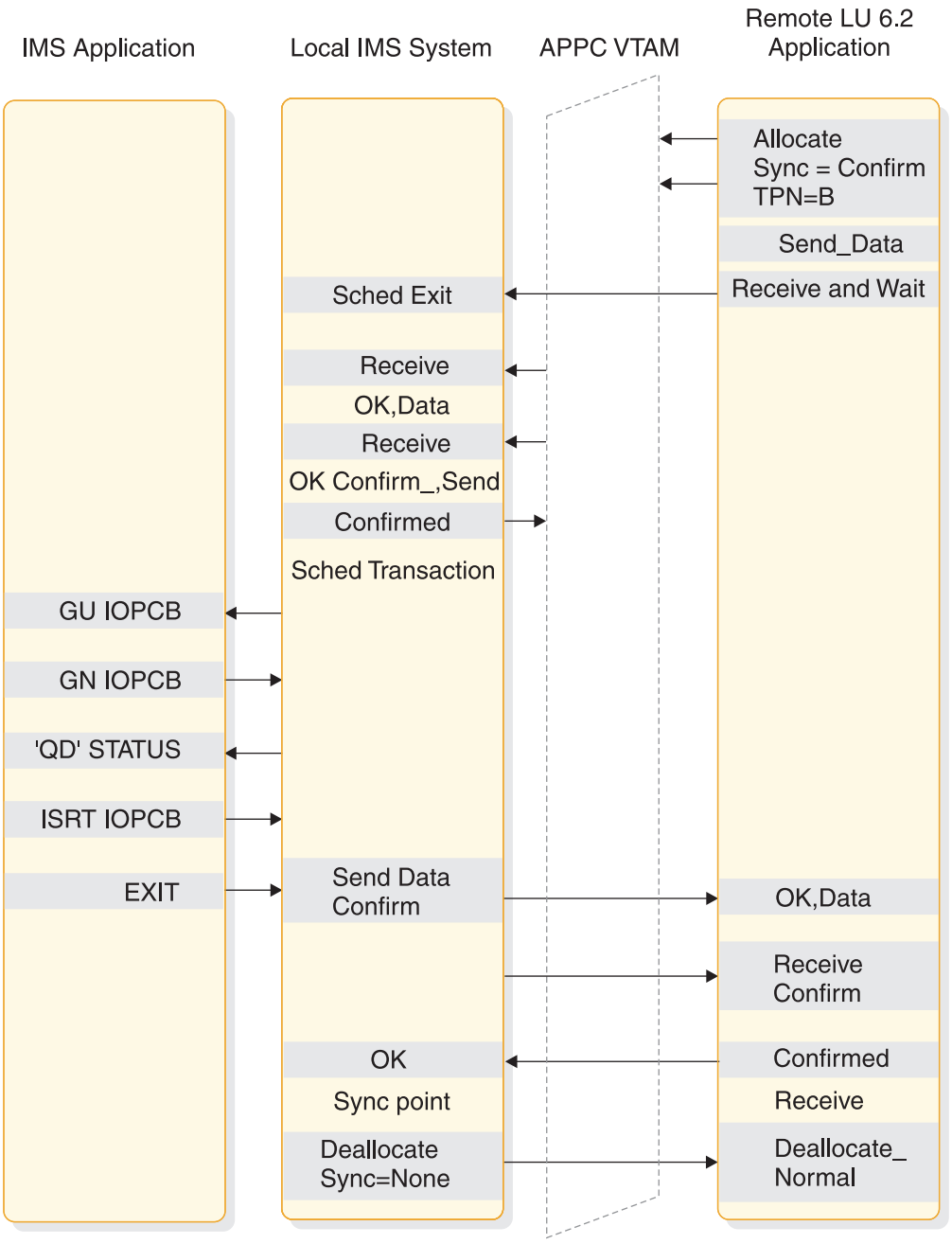


Figure 29. Flow of a local IMS synchronous transaction when Sync_level=Confirm

Figure 30 on page 125 shows the flow of a local asynchronous transaction when Sync_level is None.

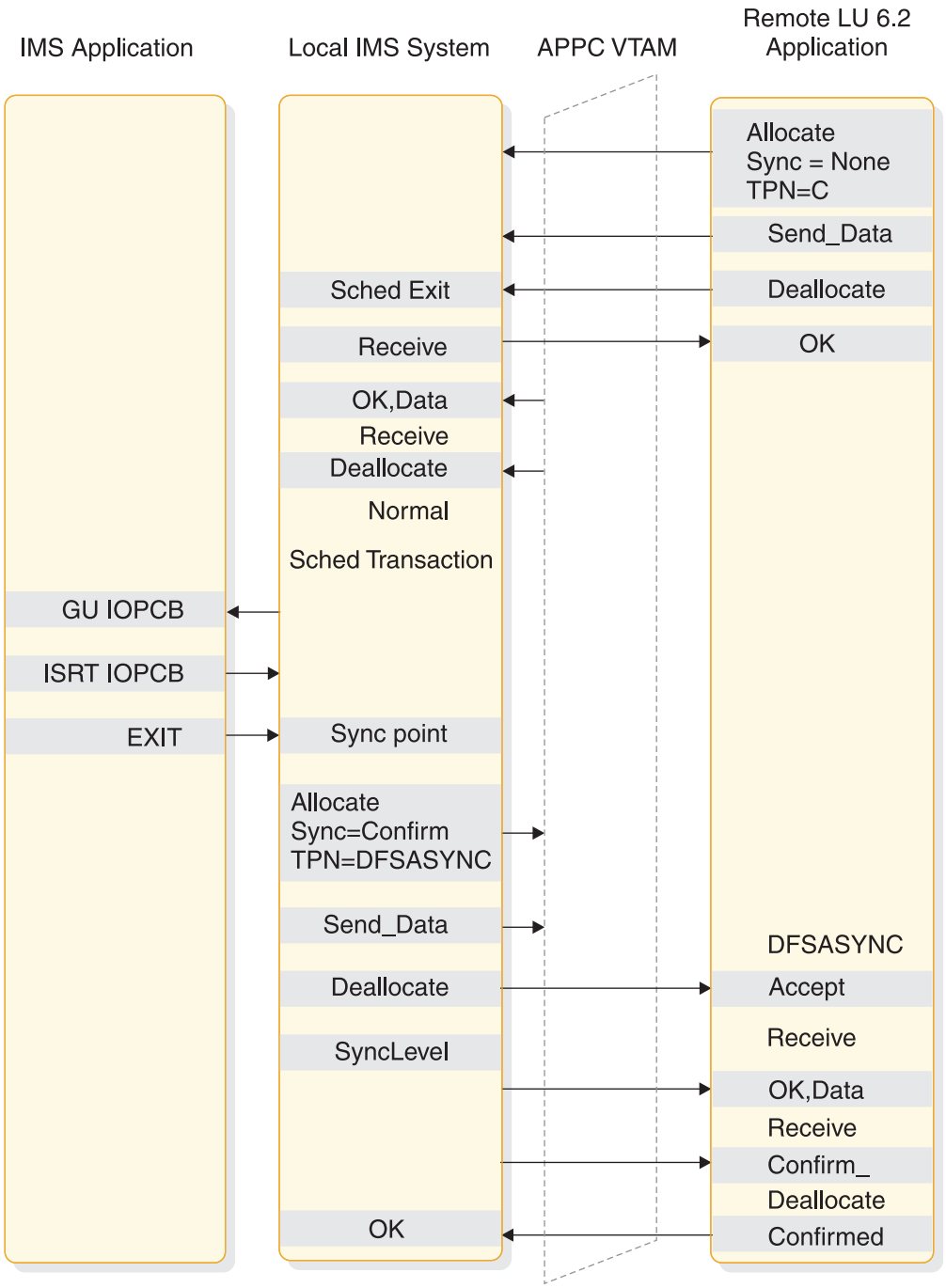


Figure 30. Flow of a local IMS asynchronous transaction when Sync_level=None

Figure 31 on page 126 shows the flow of a local asynchronous transaction when Sync_level is Confirm.

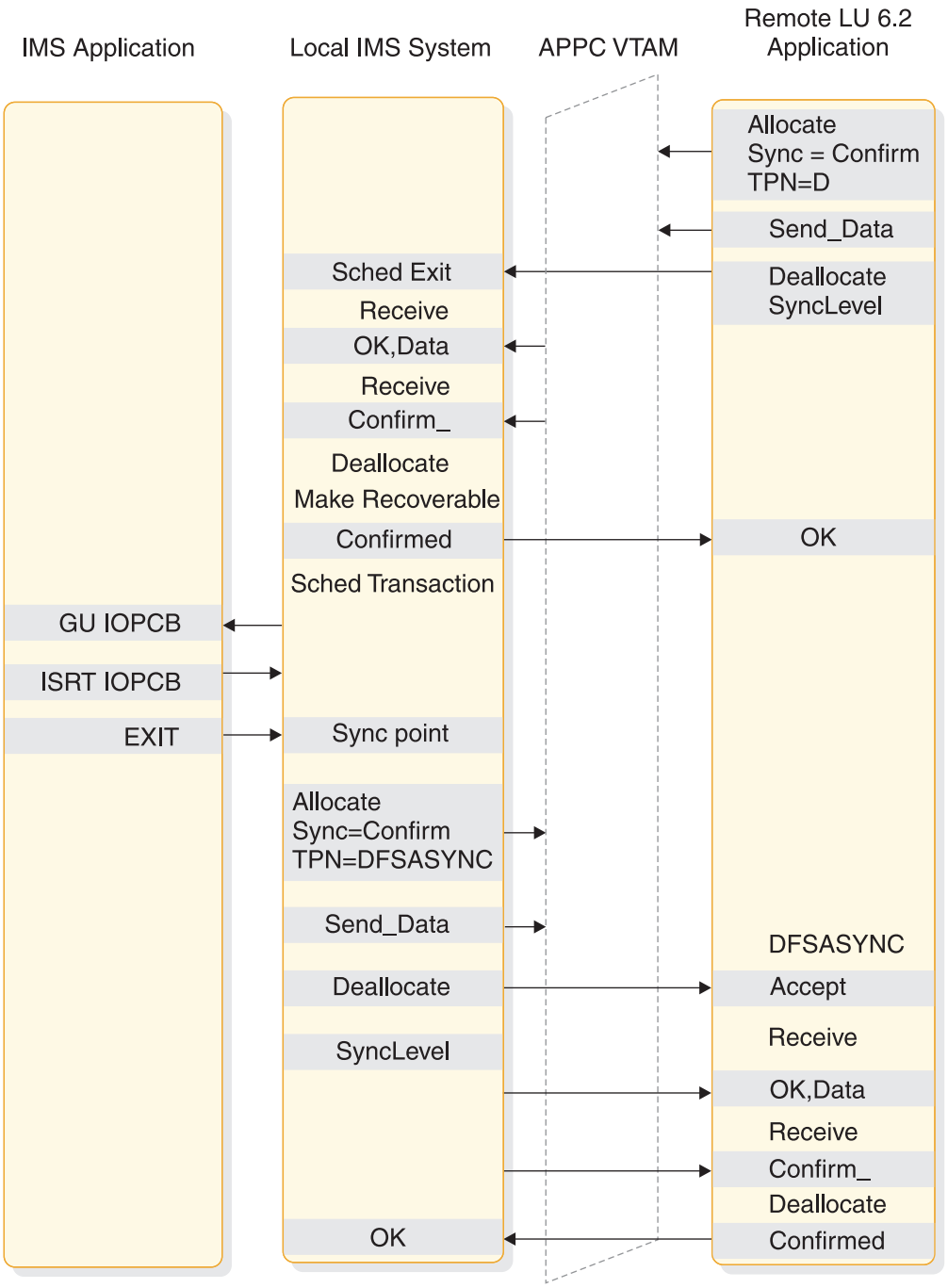


Figure 31. Flow of a local IMS asynchronous transaction when Sync_level=Confirm

The following figure shows the flow of a local conversational transaction When Sync_level is None.

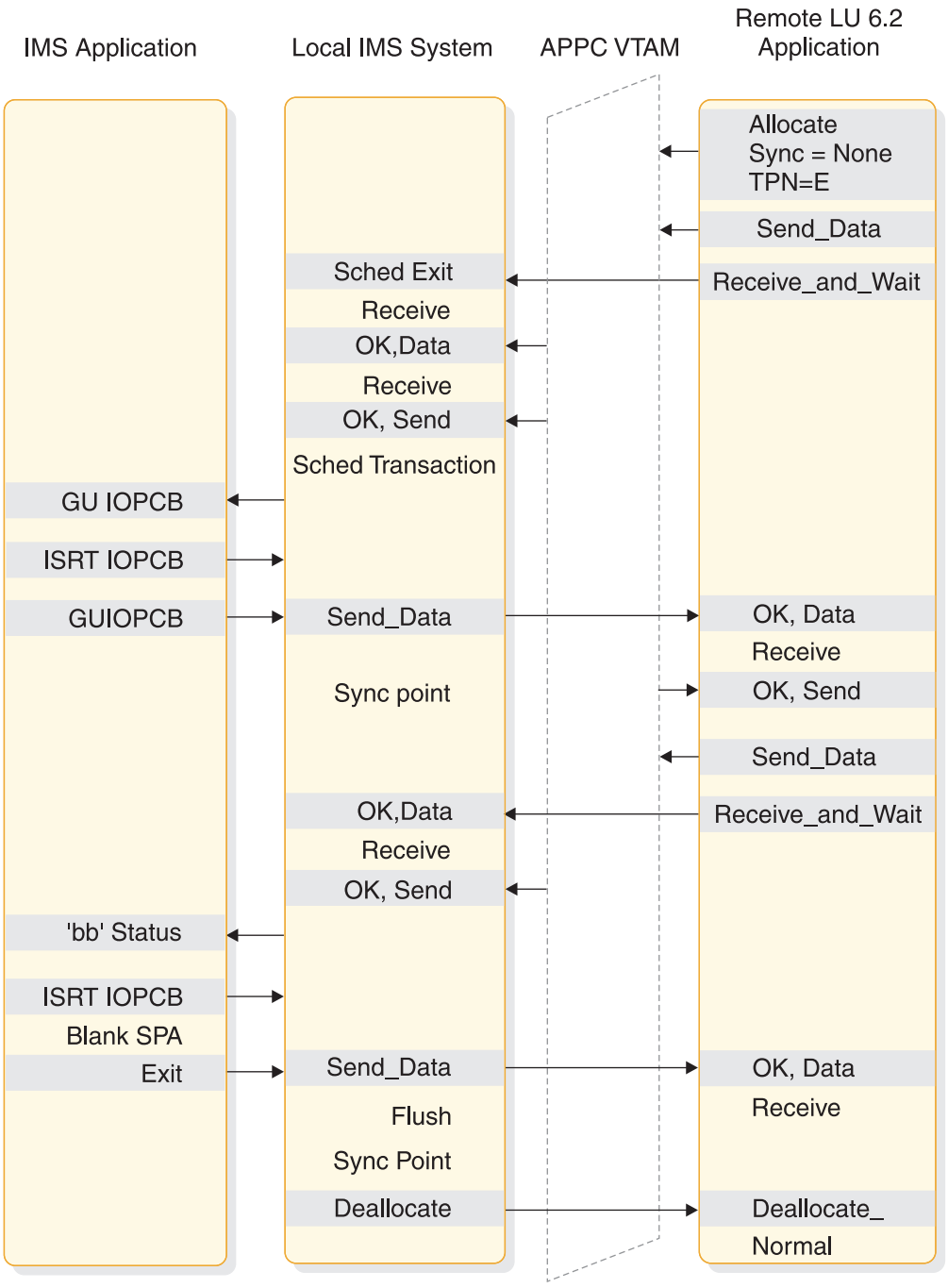


Figure 32. Flow of a local IMS conversational transaction when Sync_level=None

The following figure shows the flow of a local IMS command when Sync_level is None.

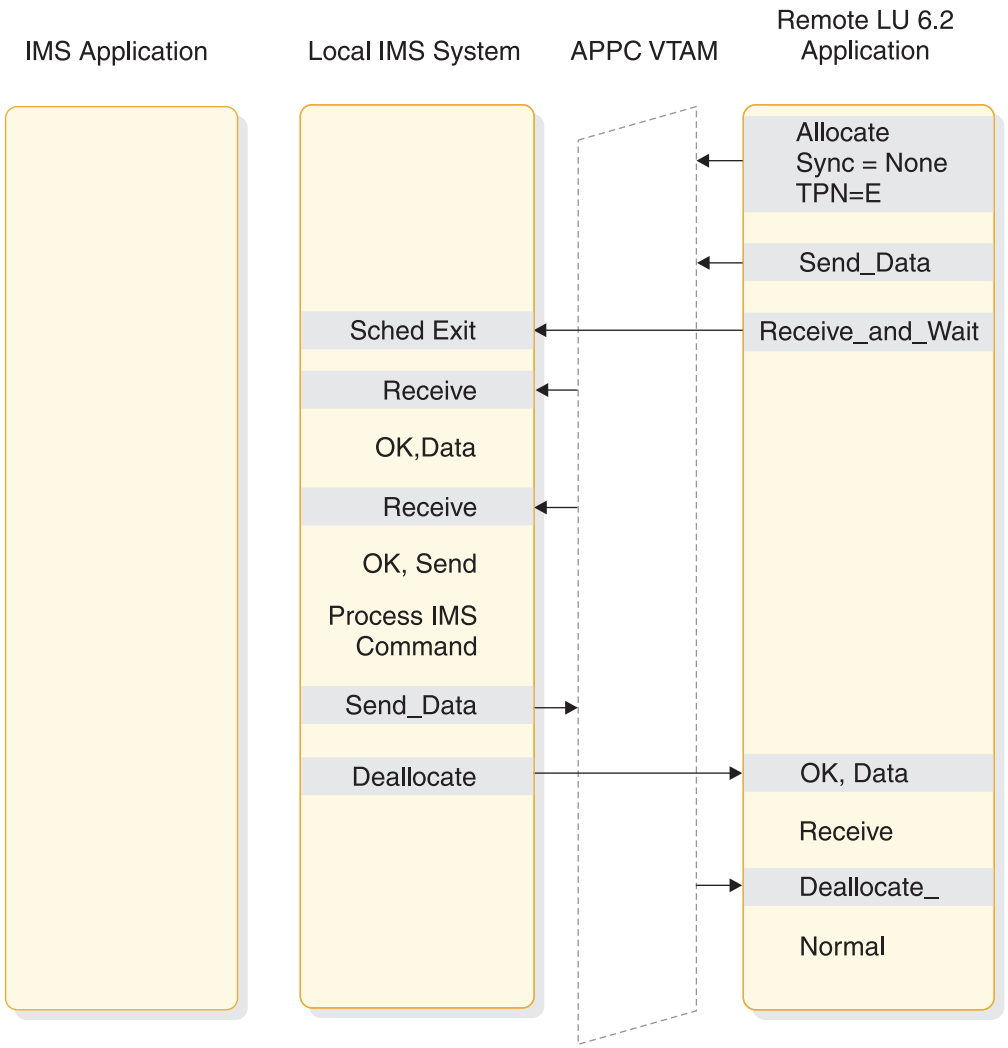


Figure 33. Flow of a local IMS command when Sync_level=None

The following figure shows the flow of a local asynchronous command when Sync_level is Confirm.

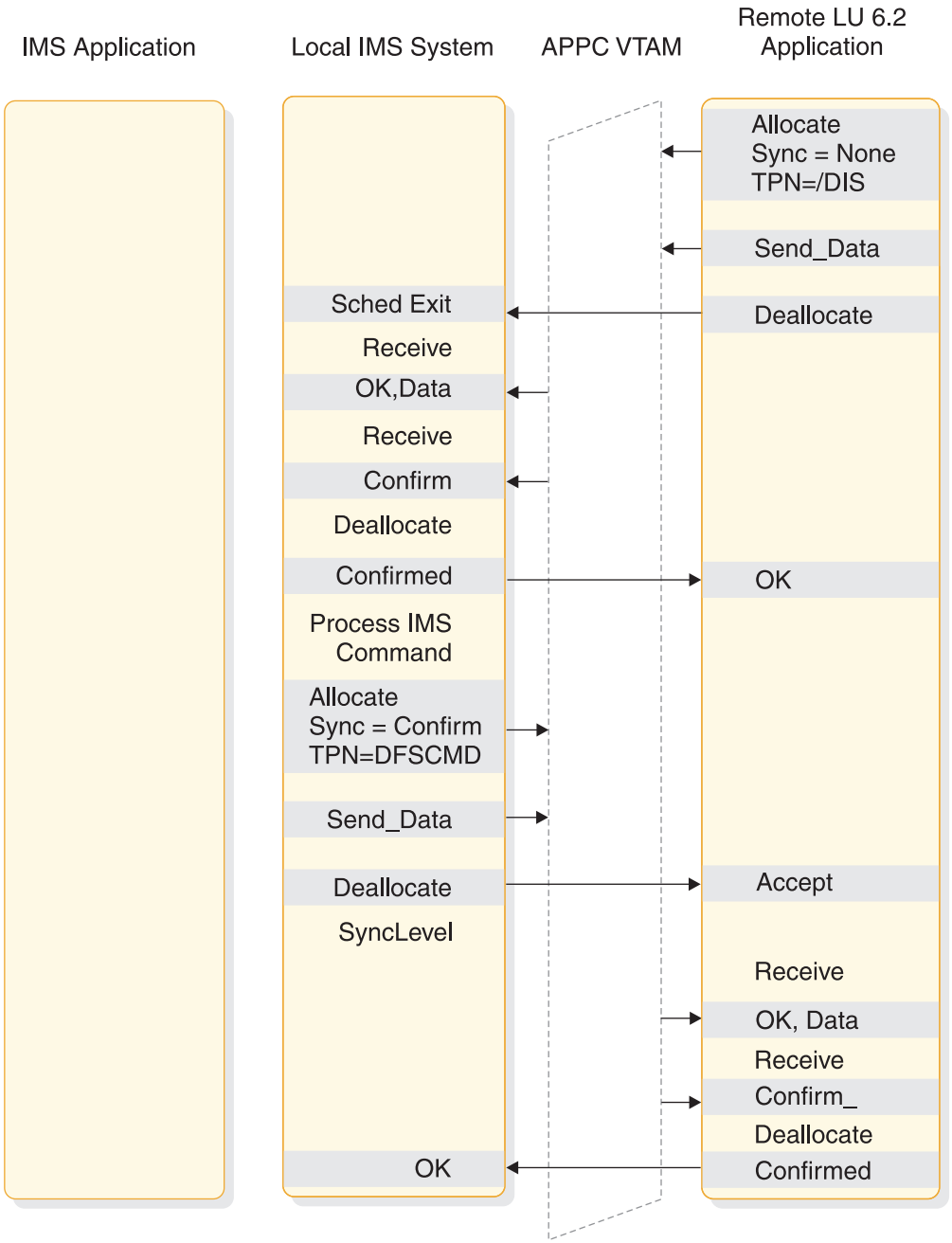


Figure 34. Flow of a local IMS asynchronous command when Sync_level=Confirm

The following figure shows the flow of a message switch When Sync_level is None.

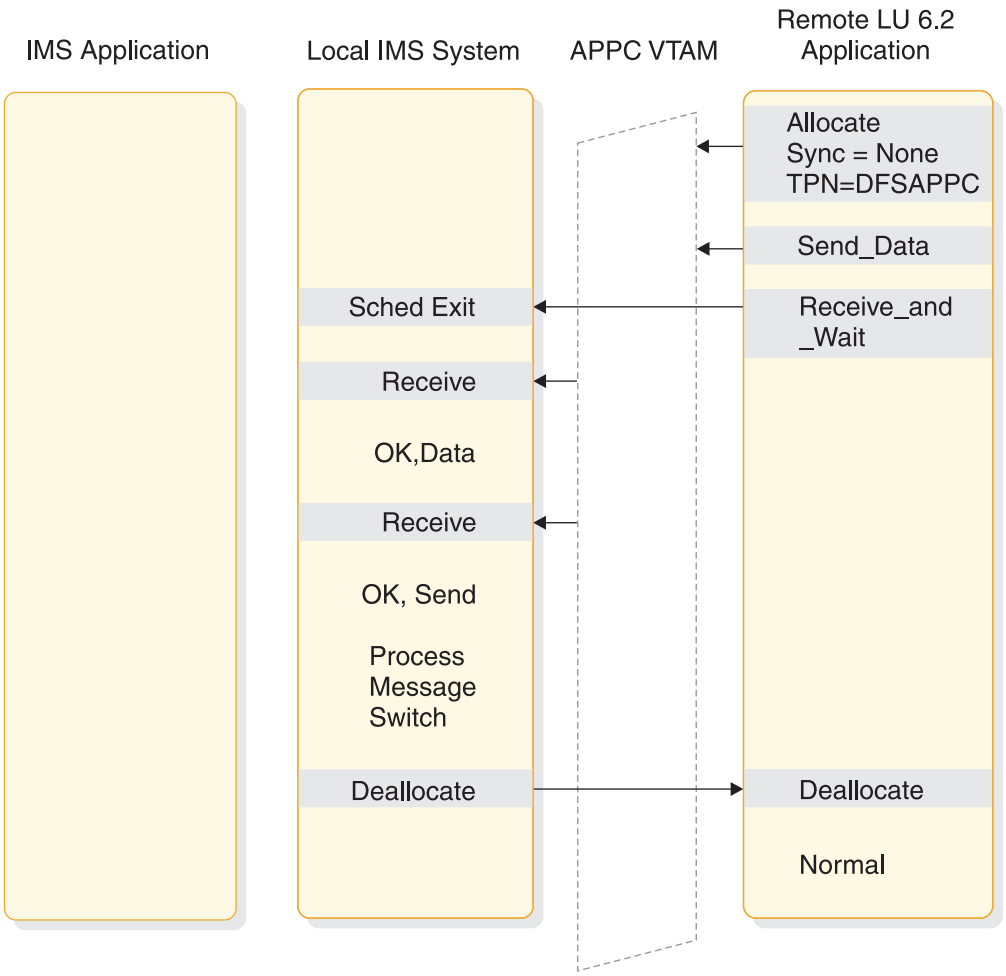


Figure 35. Flow of a message switch when Sync_level=None

Synchronous is used to verify that no error has occurred while processing DFSAPPC. If an error occurred, the error message returns before DEALLOCATE.

The following figure shows the flow of a CPI-C driven program when Sync_level is None.

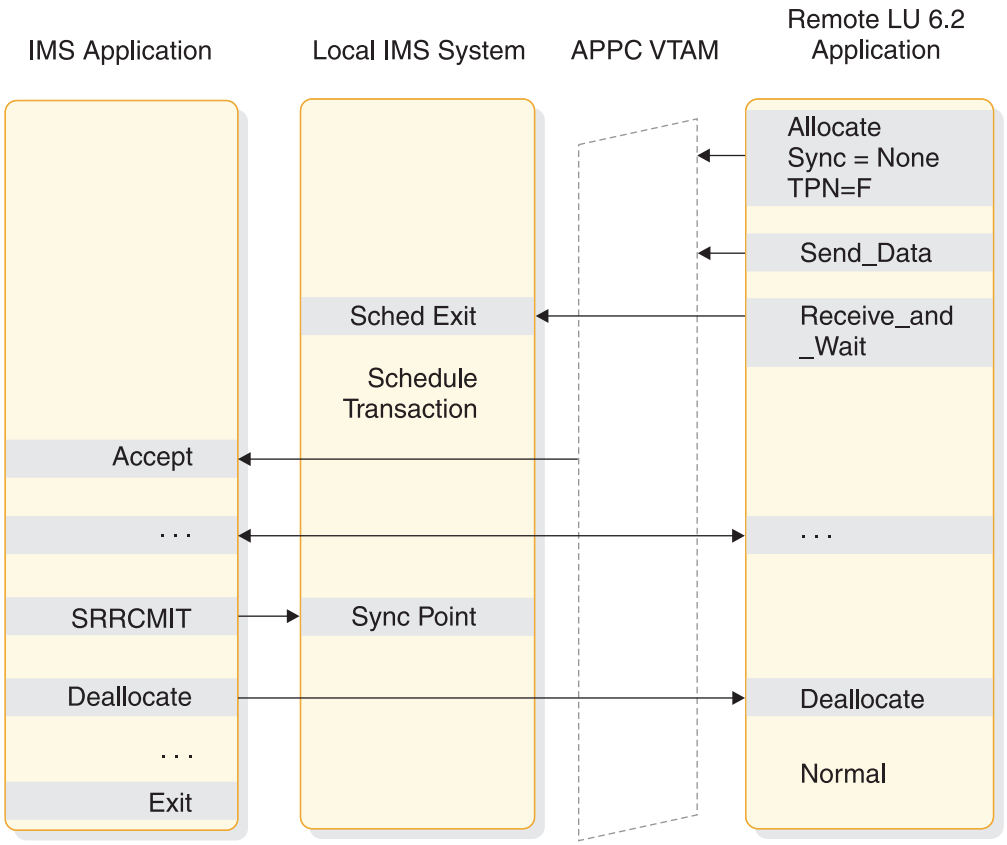


Figure 36. Flow of a local CPI communications driven program when Sync_level=None

The following figure shows the flow of a remote synchronous transaction when Sync_level is None.

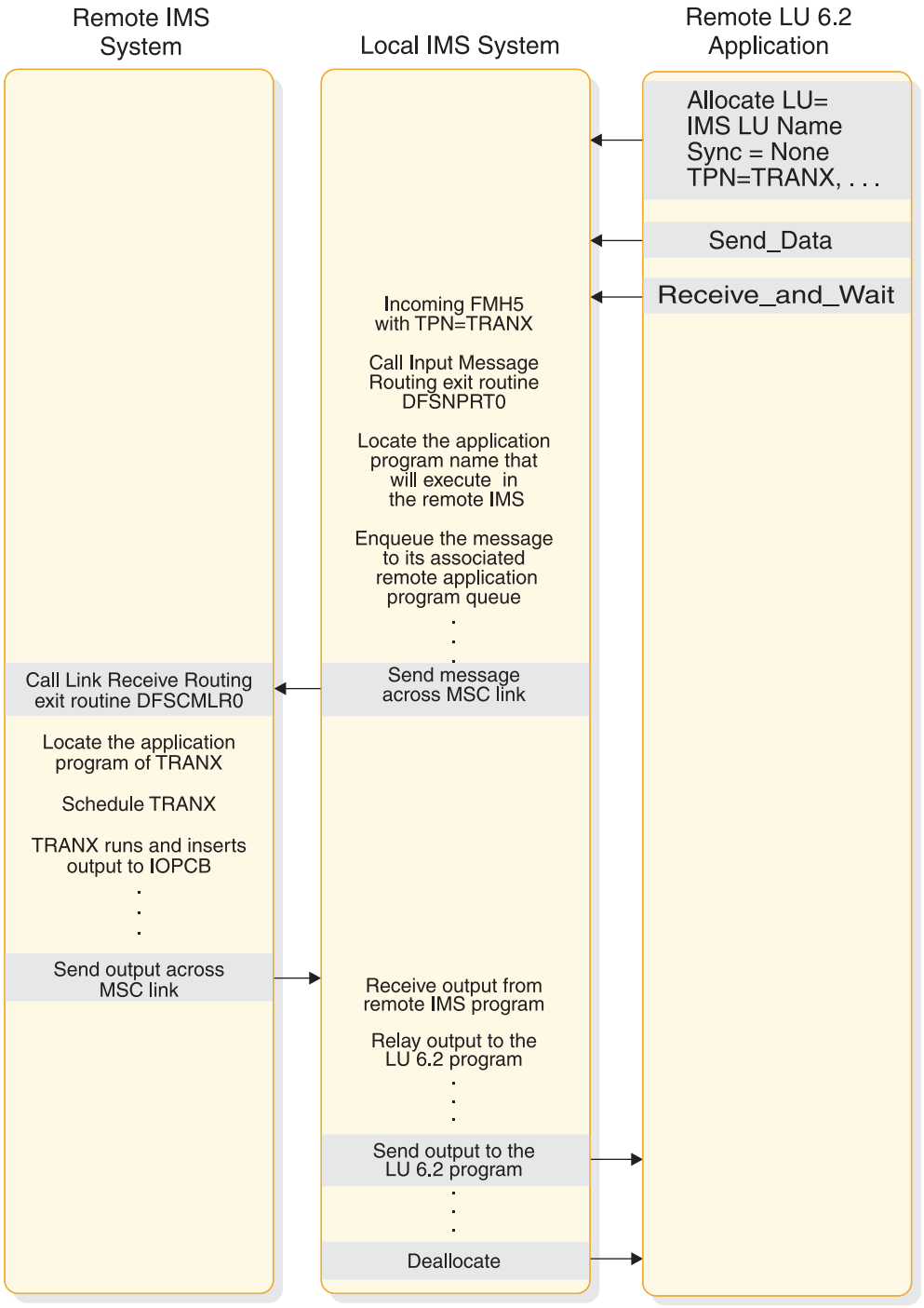


Figure 37. Flow of a remote IMS synchronous transaction when Sync_level=None

The following figure shows the flow of a remote asynchronous transaction when Sync_level is None.

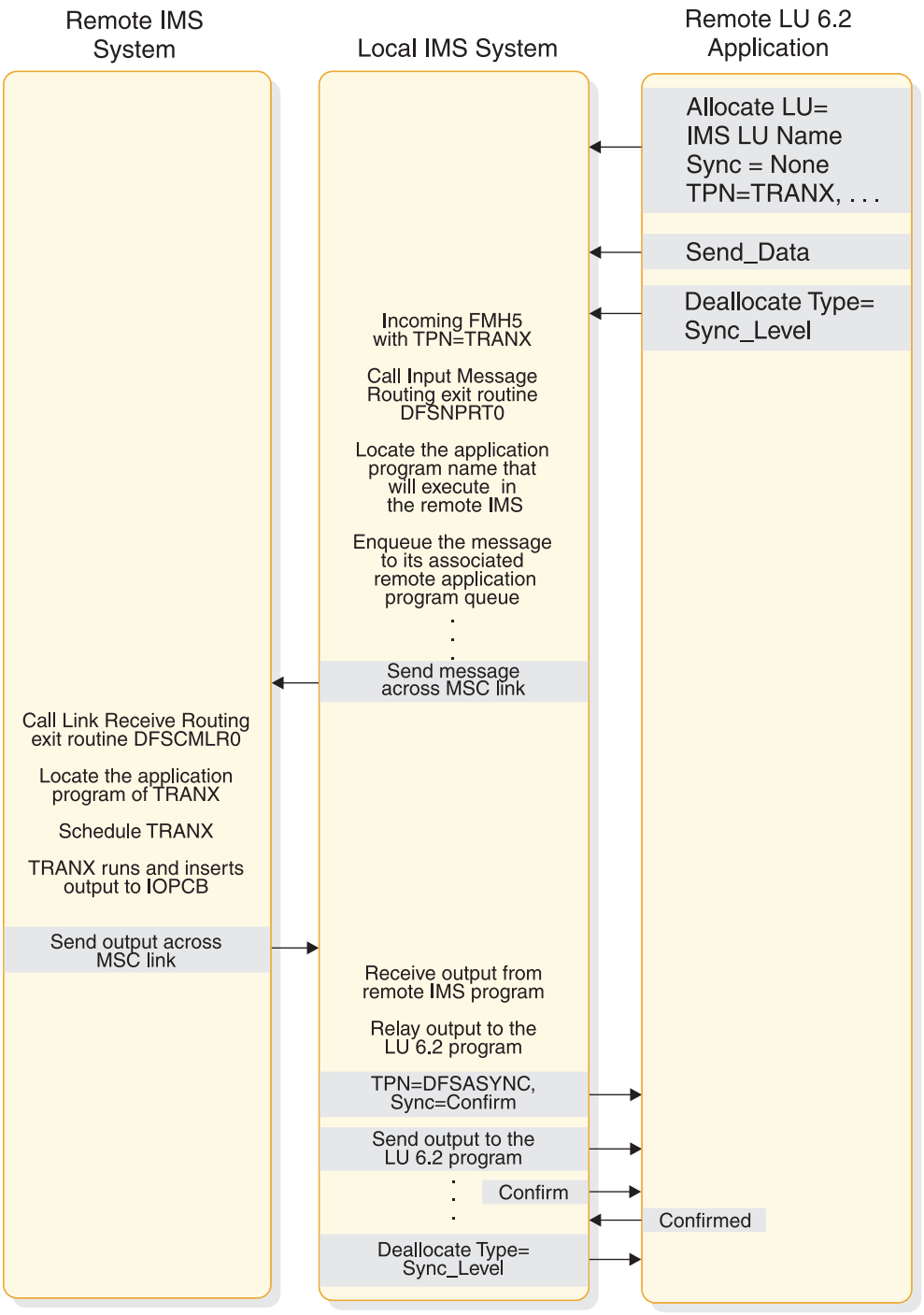


Figure 38. Flow of a remote IMS asynchronous transaction when Sync_level=None

The following figure shows the flow of a remote asynchronous transaction when Sync_level is Confirm.

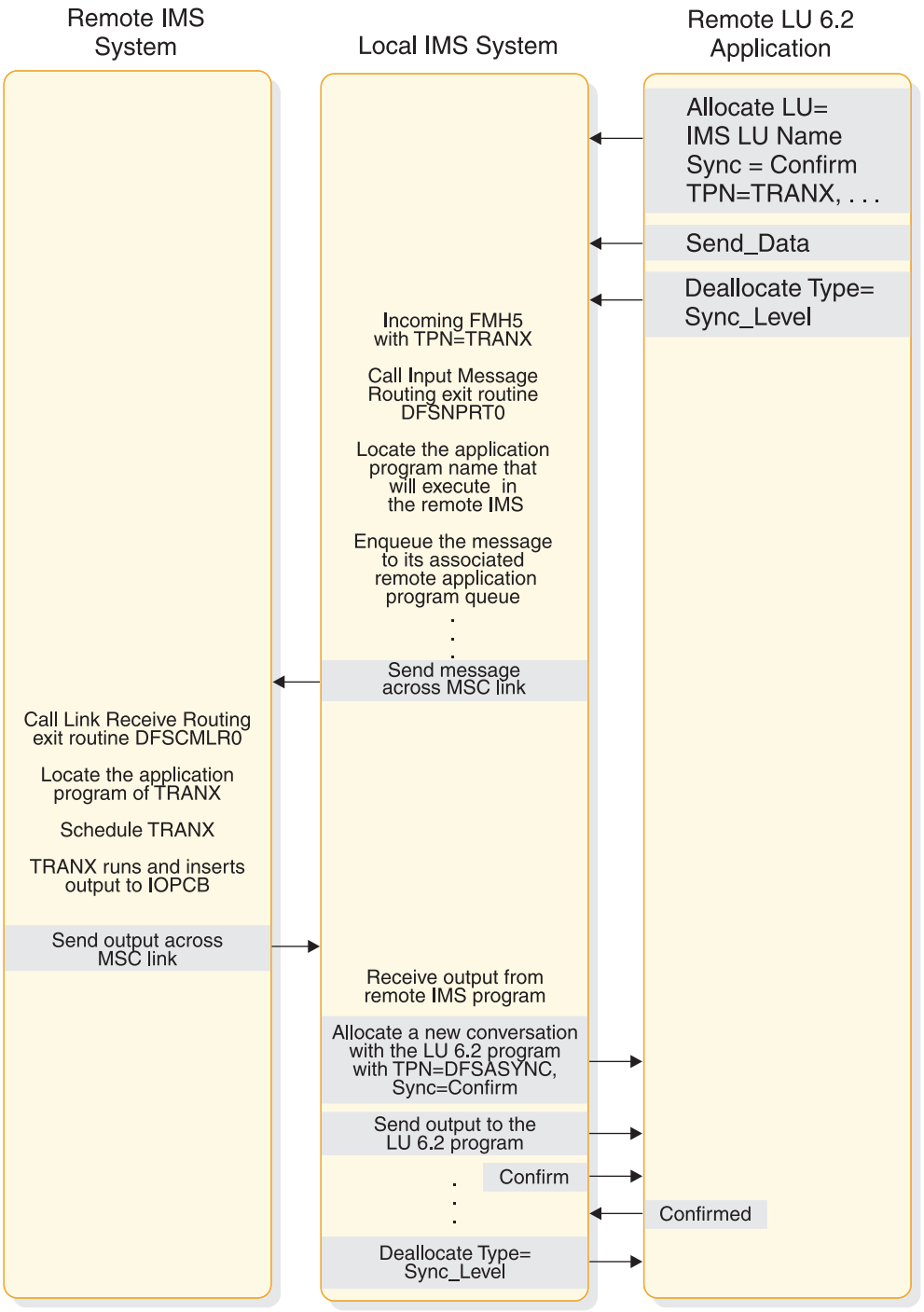


Figure 39. Flow of a remote IMS asynchronous transaction when Sync_level=Confirm

The following figure shows the flow of a remote synchronous transaction when Sync_level is Confirm.

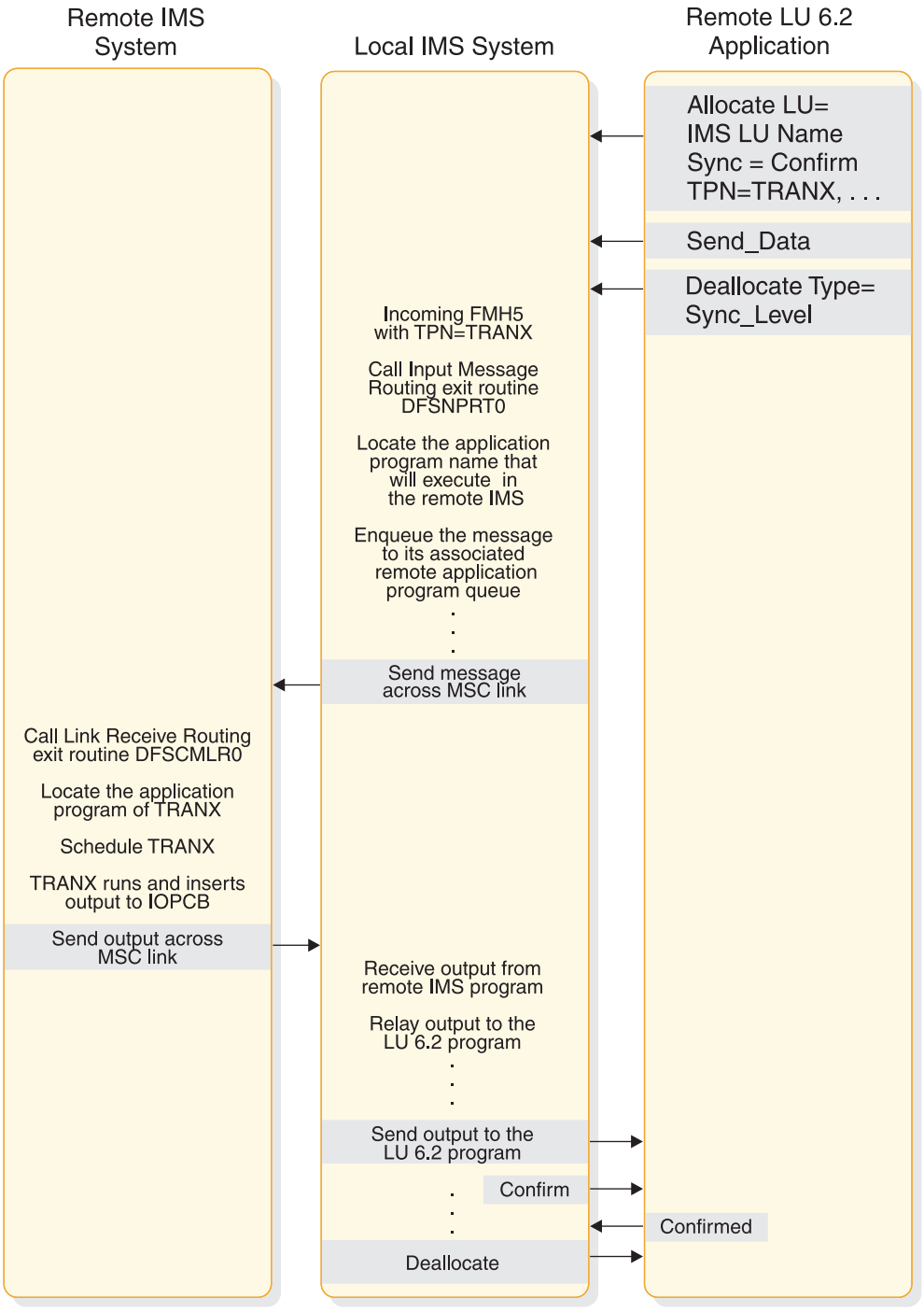


Figure 40. Flow of a remote IMS synchronous transaction when Sync_level=Confirm

The scenarios shown in the following figure provide examples of the two-phase process for the supported application program types. The LU 6.2 verbs are used to illustrate supported functions and interfaces between the components. Only parameters pertinent to the examples are included. This does not imply that other parameters are not supported.

The following figure shows a standard DL/I program commit scenario when Sync_Level=Syncpt.

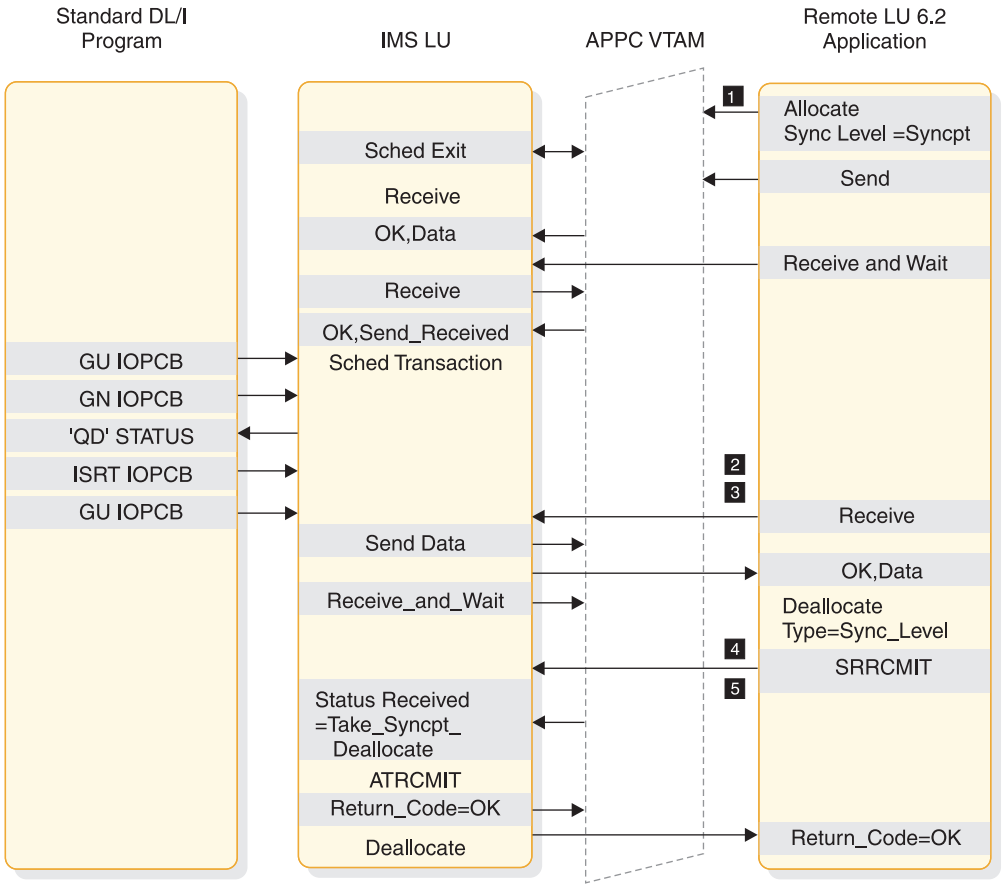


Figure 41. Standard DL/I program commit scenario when Sync_Level=Syncpt

Notes:

1. `Sync_Level=Syncpt` triggers a protected resource update.
2. This application program inserts output for the remote application to the IMS message queue.
3. The `GU` initiates the transfer of the output.
4. The remote application sends a Confirmed after receiving data (output).
5. IMS issues `ATRCMIT` (equivalent to SRRCMIT) to start the two-phase process.

The following figure shows a CPI-C driven commit scenario when Sync_Level=Syncpt.

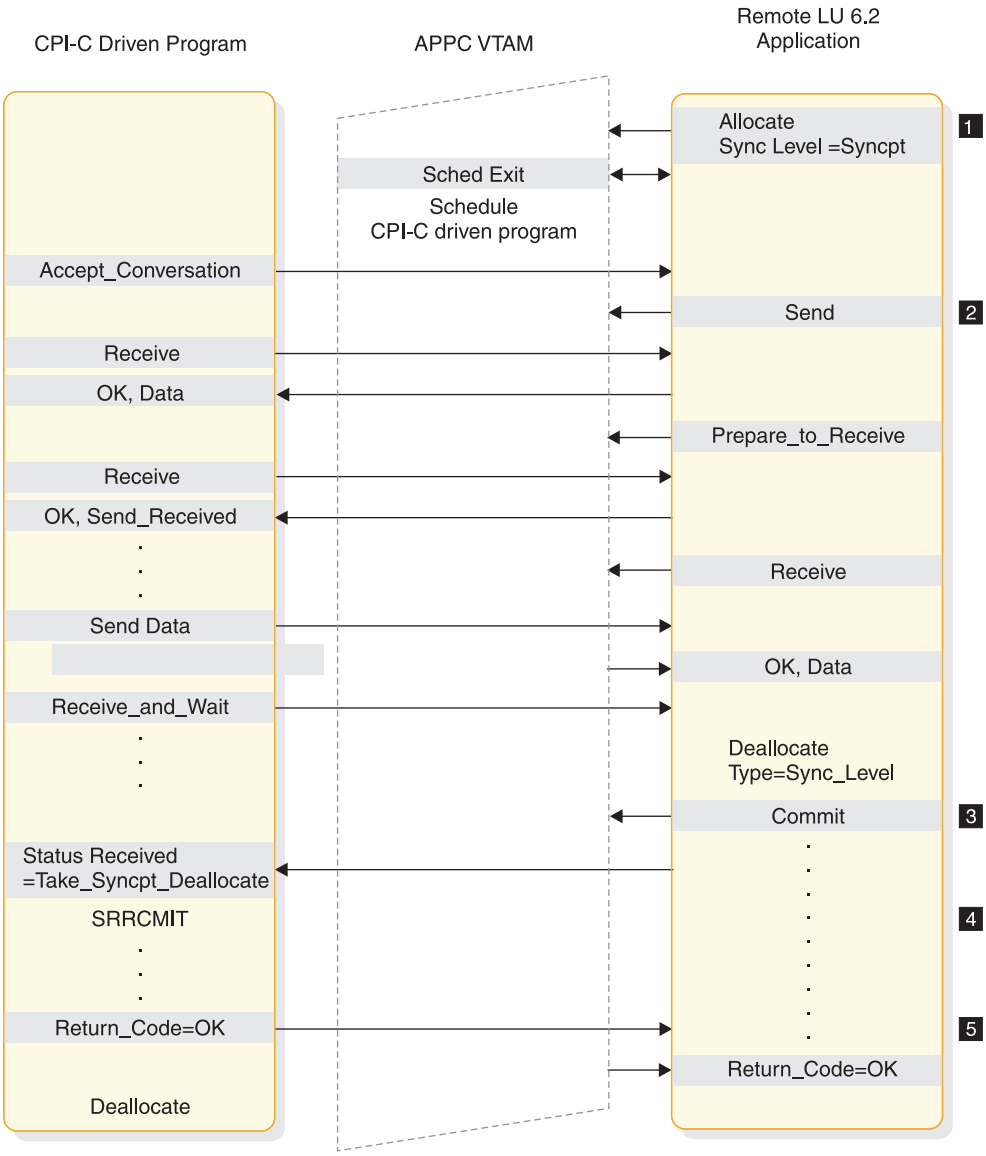


Figure 42. CPI-C driven commit scenario when Sync_Level=Syncpt

Notes:

1. `Sync_Level=Syncpt` triggers a protected resource update.
2. The programs send and receive data.
3. The remote application decides to commit the updates.
4. The CPI-C program issues `SRRCMIT` to commit the changes.
5. The commit return code is returned to the remote application.

The following figure shows a standard DL/I program backout scenario when Sync_Level=Syncpt.

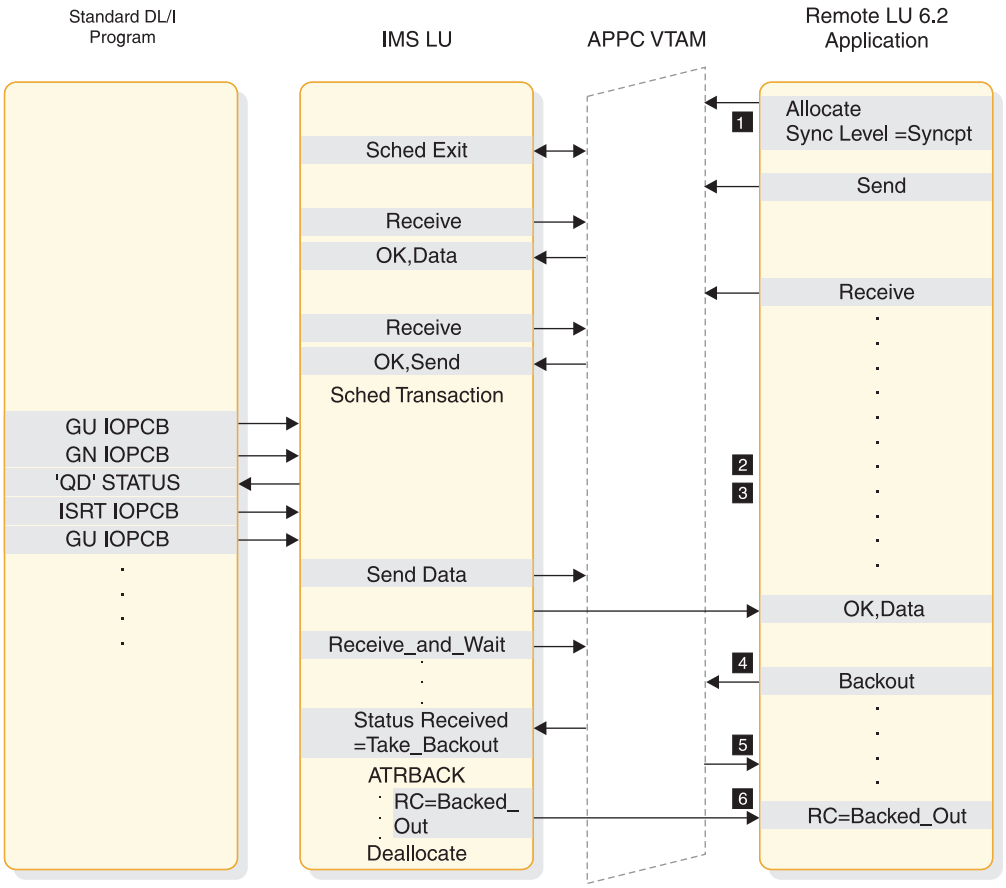


Figure 43. Standard DL/I program U119 backout scenario when Sync_Level=Syncpt

Notes:

1 `Sync_Level=Syncpt` triggers a protected-resource update.

2 This application program inserts output for the remote application to the IMS message queue.

3 The `GU` initiates the transfer of the output.

4 The remote application decides to back out any updates.

5 IMS abends the application with a U119 to back out the application.

6 The backout return code is returned to the remote application.

The following figure shows a standard DL/I program backout scenario when Sync_Level=Syncpt.

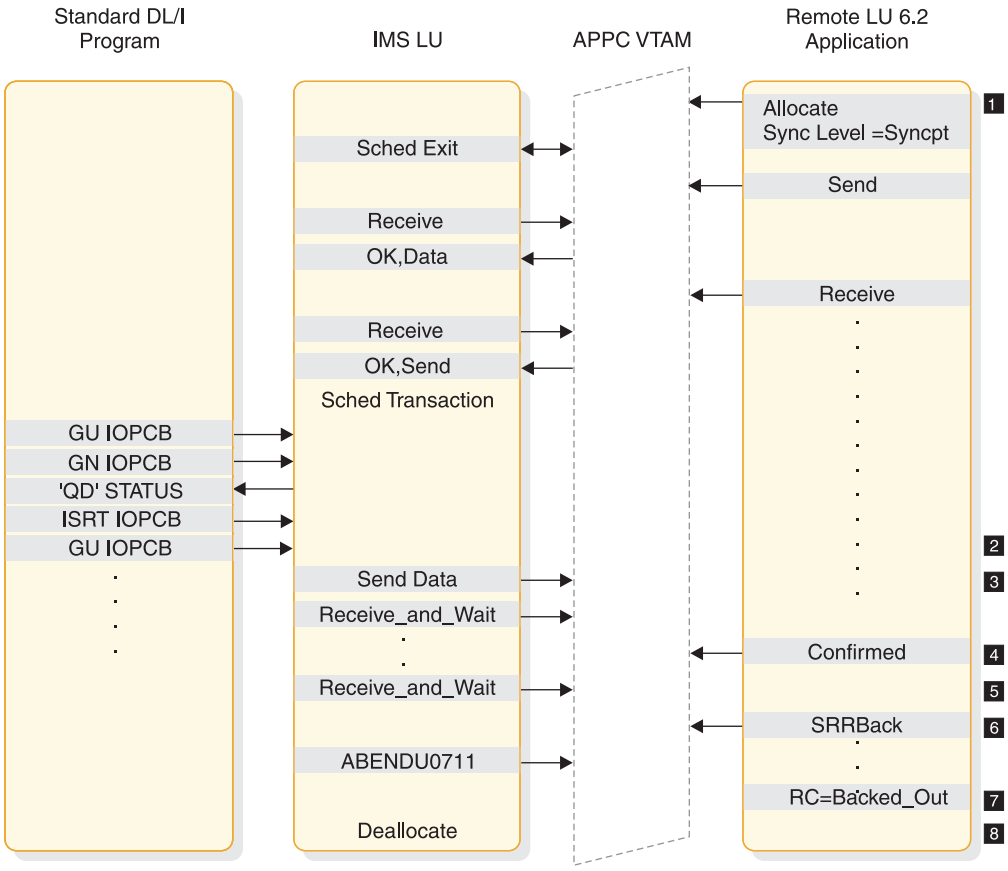


Figure 44. Standard DL/I program U0711 backout scenario when Sync_Level=Syncpt

Notes:

1. `Sync_Level=Syncpt` triggers a protected-resource update.
2. This application program inserts output for the remote application to the IMS message queue.
3. The `GU` initiates the transfer of the output.
4. The remote application sends a Confirmed after receiving data (output).
5. IMS issues `ATBRCVW on behalf of the DL/I application to wait for a commit or backout.`
6. The remote application decides to back out any updates.
7. IMS abends the application with U0711 to back out the application.
8. The backout return code is returned to the remote application.

The following figure shows a standard DL/I program ROLB scenario when Sync_Level=Syncpt.

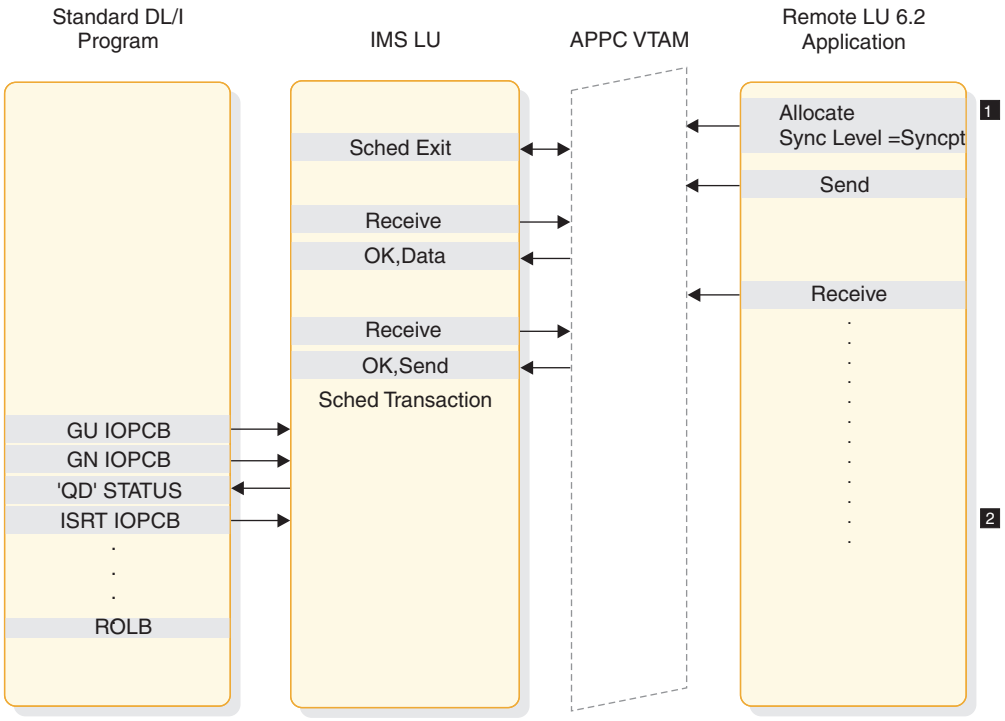


Figure 45. Standard DL/I program ROLB scenario when Sync_Level=Syncpt

Notes:

1. `Sync_Level=Syncpt` triggers a protected-resource update.
2. This application program inserts output for the remote application to the IMS message queue.

The following figure shows multiple transactions in the same commit when Sync_Level=Syncpt.

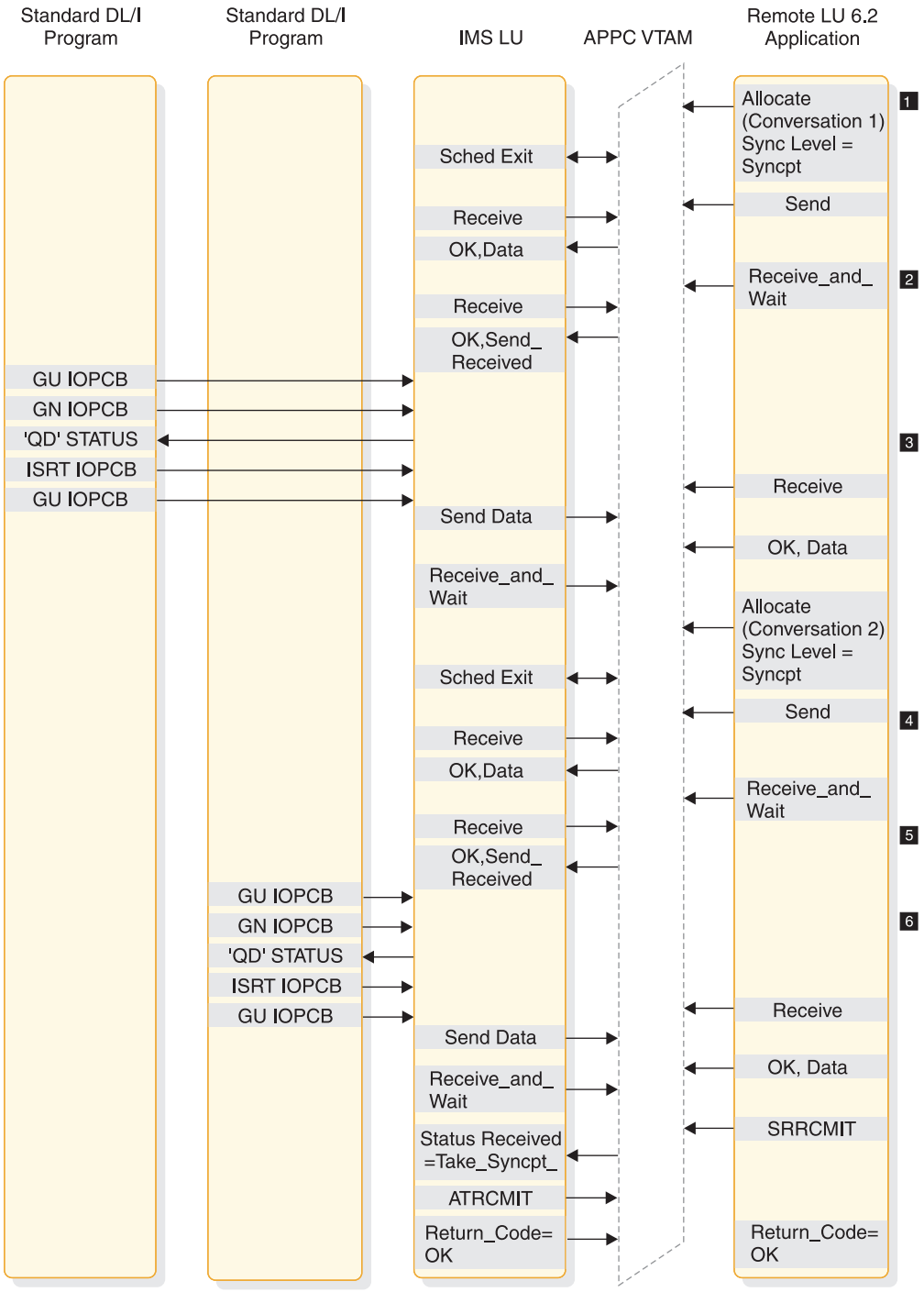


Figure 46. Multiple transactions in same commit when Sync_Level=Syncpt

Notes:

1 An allocate with `Sync_Level=Syncpt` triggers a protected resource update with Conversation 1.

2 The first transaction provides the output for Conversation 1.

3 An allocate with `Sync_Level=Syncpt` triggers a protected resource update with Conversation 2.

4 The second transaction provides the output for Conversation 2.

5 The remote application issues SRRCMIT to commit both transactions.

6 IMS issues ATRCMIT to start the two-phase process on behalf of each DL/I application.

Related concepts:

"Application objectives" on page 113

Integrity tables

The following tables show the message integrity of conversations, results of processing when integrity is compromised, and how IMS recovers APPC messages.

The following table shows the results, from the viewpoint of the IMS partner system, of normal conversation completion, abnormal conversation completion due to a session failure, and abnormal conversation completion due to non-session failures. These results apply to asynchronous and synchronous conversations and both input and output. This table also shows the outcome of the message, and the action that the partner system takes when it detects the failure. An example of an action, under "LU 6.2 Session Failure," is a programmable work station (PWS) resend.

Table 27. Message integrity of conversations

Conversation attributes	Normal	LU 6.2 session failure[1]	Other failure[2]
Synchronous Sync_level=NONE	Input: Reliable Output: Reliable	Input: PWS resend Output: PWS resend	Input: Reliable Output: Reliable
Synchronous Sync_level=CONFIRM	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable
Synchronous Sync_level=SYNCPT	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable
Asynchronous Sync_level=NONE	Input: Ambiguous Output: Reliable	Input: Undetectable Output: Reliable	Input: Undetectable Output: Reliable
Asynchronous Sync_level=CONFIRM	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable
Asynchronous Sync_level=SYNCPT	Input: Reliable Output: Reliable	Input: PWS resend Output: Reliable	Input: Reliable Output: Reliable

Notes:

1. A *session failure* is a network-connectivity breakage.
2. A *non-session failure* is any other kind of failure, such as invalid security authorization.
3. IMS resends asynchronous output if CONFIRM is lost; therefore, the PWS must tolerate duplicate output.

The following table shows the specifics of the processing windows when integrity is compromised (the message is either lost or its state is ambiguous). The table indicates the relative probability of an occurrence of each window and whether output is lost or duplicated.

A Sync_level value of NONE does not apply to asynchronous output, because IMS always uses Sync_level=CONFIRM for such output.

Table 28. Results of processing when integrity is compromised

Conversation attributes	State of window$_1$ before accepting transaction	Probability of window state	Possible action while sending response	Probability of action while sending response
Synchronous Sync_level=NONE	ALLOCATE to PREPARE_TO_ RECEIVE return	Medium	Can lose or send duplicate output.	Medium

Table 28. Results of processing when integrity is compromised (continued)

Conversation attributes	State of window[1] before accepting transaction	Probability of window state	Possible action while sending response	Probability of action while sending response
Synchronous Sync_level=CONFIRM	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small	CONFIRM to IMS receipt. Can cause duplicate output.	Small
Synchronous Sync_level=SYNCPT	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small	CONFIRM to IMS receipt. Can cause duplicate output.	Small
Asynchronous Sync_level=NONE	Allocate to Deallocate	High	CONFIRMED to IMS receipt. Can cause duplicate output.	Small
Asynchronous Sync_level=CONFIRM	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small[2]	CONFIRMED to IMS receipt. Can cause duplicate output.	Small
Asynchronous Sync_level=SYNCPT	PREPARE_TO_ RECEIVE to PREPARE_TO_ RECEIVE return	Small[2]	CONFIRMED to IMS receipt. Can cause duplicate output.	

Notes:

1. The term *window* refers to a period of time when certain events can occur, such as the consequences described in this table.
2. Can be recoverable.

The following table indicates how IMS recovers APPC transactions across IMS warm starts, XRF takeovers, APPC session failures, and MSC link failures.

Table 29. Recovering APPC messages

Message type	IMS warm start (NRE or ERE)	XRF takeover	APPC (LU 6.2) session fail	MSC LINK failure
Local Recoverable Tran., Non Resp., Non Conversation - APPC Sync. Conv. Mode - APPC Async. Conv. Mode	Discarded (2) Recovered	Discarded (4) Recovered	Discarded (6) Recovered (1)	N/A (9) N/A (9)
Local Recoverable Tran., Conv. or Resp. mode - APPC Sync. Conv. Mode - APPC Async. Conv. Mode	Discarded (2) N/A (8)	Discarded (4) N/A (8)	Discarded (6) N/A (8)	N/A (9) N/A (8,9)
Local Non Recoverable Tran., - APPC Sync. Conv. Mode - APPC Async. Conv. Mode	Discarded (2) Discarded (2)	Discarded (4)	Discarded (6) Recovered (1)	N/A (9) N/A (9)
Remote Recoverable Tran., Non Resp., Non Conv. - APPC Sync. Conv. Mode - APPC Async. Conv. Mode	Discarded (2,5) Recovered	Discarded (3,5) Recovered	Recovered (1) Recovered (1)	Recovered (7) Recovered (7)
Remote Recoverable Tran., Conv. or Resp. mode - APPC Sync. Conv. Mode - APPC Async. Conv. Mode	Discarded (2,5) N/A (8)	Discarded (3,5) N/A (8)	Recovered (1) N/A (8)	Recovered (7) N/A (8)

Table 29. Recovering APPC messages (continued)

Message type	IMS warm start (NRE or ERE)	XRF takeover	APPC (LU 6.2) session fail	MSC LINK failure
Remote Non Recoverable Tran., -				
APPC Sync. Conv. Mode - APPC	Discarded (2,5)	Discarded (3,5)	Recovered (1)	Recovered (7)
Async. Conv. Mode	Discarded (2,5)	Discarded (3,5)	Recovered (1)	Recovered (7)

Note:

1. This recovery scenario assumes the message was enqueued before failure; otherwise, the message is discarded.
2. The message is discarded during IMS warm-start processing.
3. The message is discarded when the MSC link is restarted and when the message is taken off the queue (for sending across the link).
4. The message is discarded when the message region is started and when the message is taken off the queue (for processing by the application program).
5. For all remote MSC APPC transactions, if the message has already been sent across the MSC link to the remote system when the failure occurs in the local IMS, the message is processed. After the message is processed by the remote application program and a response message is sent back to the local system, it is enqueued to the DFSASYNC TP name of the LU 6.2 device or program that submitted the original transaction.
6. At sync point, the User Message Control Error exit routine (DFSCMUX0) can prevent the transaction from being aborted and the output message can be rerouted (recovered).

   For more information about this exit routine, see *IMS Version 11 Exit Routines*.
7. The standard MSC Link recovery protocol recovers all messages that are queued or are in the process of being sent across the MSC link when the link fails.
8. IMS conversational-mode and response-mode transactions cannot be submitted from APPC asynchronous conversation sessions. APPC synchronous conversation-mode must be used.
9. MSC link failures do not affect local transactions.

DFSAPPC message switch

DFSAPPC is an LU 6.2 descriptor that provides an IMS system service.

It allows LU 6.2 application programs to send messages to the following:

- Application programs (transactions)
- IMS-managed local or remote LTERMs (message switches)
- LU name and TP name

Messages sent with the LTERM= option are directed to IMS-managed local or remote LTERMs. Messages sent without the LTERM= option are sent to the appropriate LU 6.2 application or IMS application program.

Because the LTERM can be an LU 6.2 descriptor name, the message is sent to the LU 6.2 application program as if an LU 6.2 device had been explicitly selected.

With DFSAPPC, message delivery is asynchronous. If a message is allocated and the allocate fails, the message is held on the IMS message queue until it can be successfully delivered.

Example: In the LU 6.2 conversation example, an IMS application issues a DFSAPPC message switch to its partner with the LU name FRED and TPN name REPORT. REPI is the user data.

```
DFSAPPC  (TPN=REPORT LU=FRED) REP1
```

You can use a 17-byte network-qualified name in the LU= field.

Restriction: LU 6.2 architecture prohibits the use of the ALTRESP PCB on a `CHNG` call in an LU 6.2 conversation. The LU 6.2 conversation can only be associated with the IOPCB. The application sends a message on the existing LU 6.2 conversation (synchronous) or has IMS create a new conversation (asynchronous) using the IOPCB. Since there is no LTERM associated with an LU 6.2 conversation, only the IOPCB represents the original LU 6.2 conversation.

Related Reading: For more information about DFSAPPC, see *IMS Version 11 Communications and Connections*.

Chapter 8. Testing an IMS application program

You should perform a program unit test on your IMS application program to ensure that the program correctly handles its input data, processing, and output data. The amount and type of testing you do depends on the individual program.

Recommendations for testing an IMS program

Before you start testing your program, be aware of your established test procedures.

To start testing, you need the following three items:

- Test JCL.
- A test database. Never test a program using a production database because the program, if faulty, might damage valid data.
- Test input data. The input data that you use need not be current, but it should be valid. You cannot be sure that your output data is valid unless you use valid input data.

The purpose of testing the program is to make sure that the program can correctly handle all the situations that it might encounter. To thoroughly test the program, try to test as many of the paths that the program can take as possible.

Recommendations:

- Test each path in the program by using input data that forces the program to execute each of its branches.
- Be sure that your program tests its error routines. Again, use input data that will force the program to test as many error conditions as possible.
- Test the editing routines your program uses. Give the program as many different data combinations as possible to make sure it correctly edits its input data.

Testing DL/I call sequences (DFSDDLT0) before testing your IMS program

The DL/I test program, DFSDDLT0, is an IMS application program that executes the DL/I calls you specify against any database.

Restriction: DFSDDLT0 does not work if you are using a coordinator controller (CCTL).

An advantage of using DFSDDLT0 is that you can test the DL/I call sequence you will use prior to coding your program. Testing the DL/I call sequence before you test the program makes debugging easier, because by the time you test the program, you know that the DL/I calls are correct. When you test the program, and it does not execute correctly, you know that the DL/I calls are not part of the problem if you have already tested them using DFSDDLT0.

For each DL/I call that you want to test, you give DFSDDLT0 the call and any SSAs that you are using with the call. DFSDDLT0 then executes and gives you the results of the call. After each call, DFSDDLT0 shows you the contents of the DB PCB mask and the I/O area. This means that for each call, DFSDDLT0 checks the

© Copyright IBM Corp. 1974, 2015

access path you have defined for the segment, and the effect of the call. DFSDDLT0 is helpful in debugging because it can display IMS application control blocks.

To indicate to DFSDDLT0 the call you want executed, you use four types of control statements:

Status statements establish print options for DFSDDLT0's output and select the DB PCB to use for the calls you specify.

Comment statements let you choose whether you want to supply comments.

Call statements indicate to DFSDDLT0 the call you want to execute, any SSAs you want used with the call, and how many times you want the call executed.

Compare statements tell DFSDDLT0 that you want it to compare its results after executing the call with the results you supply.

In addition to testing call sequences to see if they work, you can also use DFSDDLT0 to check the performance of call sequences.

Using BTS to test your IMS program

IMS Batch Terminal Simulator for z/OS (BTS) is a valuable tool for testing programs because you can use it to test call sequences. The documentation that BTS produces is helpful in debugging. You can also test online application programs without actually running them online.

Restriction: BTS does not work if you are using a CCTL or running under DBCTL.

Related reading: For information about how to use BTS, see *IMS Batch Terminal Simulator for z/OS User's Guide*.

Tracing DL/I calls with image capture for your IMS program

The DL/I image capture program (DFSDLTR0) is a trace program that can trace and record DL/I calls issued by all types of IMS application programs.

Restriction: The image capture program does not trace calls to Fast Path databases.

You can run the image capture program in a DB/DC or a batch environment to:

- **Test your program**

  If the image capture program detects an error in a call it traces, it reproduces as much of the call as possible, although it cannot document where the error occurred, and cannot always reproduce the full SSA.
- **Produce input for DFSDDLT0**

  You can use the output produced by the image capture program as input to DFSDDLT0. The image capture program produces status statements, comment statements, call statements, and compare statements for DFSDDLT0.
- **Debug your program**

  When your program terminates abnormally, you can rerun the program using the image capture program, which can then reproduce and document the conditions that led to the program failure. You can use the information in the report produced by the image capture program to find and fix the problem.

Using image capture with DFSDDLT0

The image capture program produces the following control statements that you can use as input to DFSDDLT0.

- **Status statements**

  When you invoke the image capture program, it produces the status statement. The status statement it produces:

  – Sets print options so that DFSDDLT0 prints all call trace comments, all DL/I calls, and the results of all comparisons.

  – Determines the new relative PCB number each time a PCB change occurs while the application program is executing.

- **Comments statement**

  The image capture program also produces a comments statement when you invoke it. The comments statements give:

  – The time and date IMS started the trace

  – The name of the PSB being traced

  The image capture program also produces a comments statement preceding any call in which IMS finds an error.

- **Call statements**

  The image capture program produces a call statement for each DL/I call the application program issues. It also generates a `CHKP` call when it starts the trace and after each commit point or `CHKP` request.

- **Compare statements**

  The image capture program produces data and PCB comparison statements if you specify COMP on the `TRACE` command (if you run the image capture program online), or on the DLITRACE control statement (if you run the image capture program as a batch job).

Restrictions on using image capture output

The status statement of the image capture call is based on relative PCB position.

When the PCB parameter LIST=NO has been specified, the status statement may need to be changed to select the PCB as follows:

- If all PCBs have the parameter LIST=YES, the status statement does not need to be changed.
- If all PCBs have the parameter LIST=NO, the status statement needs to be changed from the relative PCB number to the correct PCB name.
- If some PCBs have the parameter LIST=YES and some have the parameter LIST=NO, the status statement needs to be changed as follows:

  – The PCB relative position is based on all PCBs as if LIST=YES.

  – For PCBs that have a PCB name, the status statement can be changed to use the PCB name based on a relative PCB number.

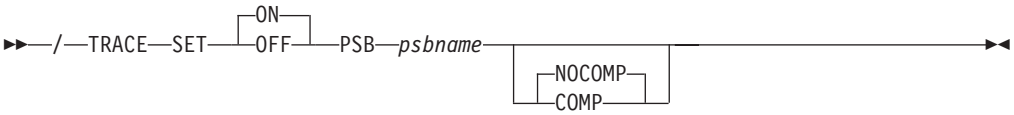
  – For PCBs that have LIST=YES and no PCB name, change the relative PCB number to refer to the relative PCB number in the user list by looking at the PCB list using LIST=YES and LIST=NO.

Running image capture online

When you run the image capture program online, the trace output goes to the IMS log data set. To run the image capture program online, you issue the IMS `TRACE` command from the IMS master terminal.

If you trace a BMP or an MPP and you want to use the trace results with DFSDDLT0, the BMP or MPP must have exclusive write access to the databases it processes. If the application program does not have exclusive access, the results of DFSDDLT0 may differ from the results of the application program. When you trace a BMP that accesses GSAM databases, you must include an //IMSERR DD statement to get a formatted dump of the GSAM control blocks.

The following diagram shows the TRACE command format:



SET ON|OFF
Turns the trace on or off.

PSB *psbname*
Specifies the name of the PSB you want to trace. You can trace more than one PSB at the same time by issuing a separate TRACE command for each PSB.

COMP|NOCOMP
Specifies whether you want the image capture program to produce data and PCB compare statements to be used as input to DFSDDLT0.

Running image capture as a batch job

To run the image capture program as a batch job, you use the DLITRACE control statement in the DFSVSAMP DD data set.

In the DLITRACE control statement, you specify:

- Whether you want to trace all of the DL/I calls the program issues or trace only a certain group of calls.
- Whether you want the trace output to go to:
  - A sequential data set that you specify
  - The IMS log data set
  - Both sequential and IMS log data sets

If the program being traced issues CHKP and XRST calls, the checkpoint and restart information may not be directly reproducible when you use the trace output with DFSDDLT0.

When you run DFSDDLT0 in an IMS DL/I or DBB batch region with trace output, the results are the same as the application program's results, but only if the database has not been altered.

For information on the format of the DLITRACE control statement in the DFSVSAMP DD data set, see the topic “Defining DL/I call image trace” in *IMS Version 11 System Definition*.

Retrieving image capture data from the log data set

If the trace output is sent to the IMS log data set, you can retrieve it by using utility DFSERA10 and a DL/I call trace exit routine, DFSERA50. DFSERA50 deblocks, formats, and numbers the image capture program records that are to be retrieved.

To use DFSERA50, you must insert a DD statement defining a sequential output data set in the DFSERA10 input stream. The default ddname for this DD statement is TRCPUNCH. The statement must specify BLKSIZE=80.

For example, you can use the following examples of DFSERA10 input control statements in the SYSIN data set to retrieve the image capture program data from the log data set:

- **Print all image capture program records:**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,FLDTYP=X
```

- **Print selected image capture program records by PSB name:**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT OFFSET=25,VLDTYP=C,FLDLEN=8,
               VALUE=psbname, COND=E
```

- **Format image capture program records (in a format that can be used as input to DFSDDLT0):**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT EXITR=DFSERA50,OFFSET=25,FLDTYP=C
               VALUE=psbname,FLDLEN=8,DDNAME=OUTDDN,COND=E
```

Remember: The DDNAME= parameter names the DD statement to be used by DFSERA50. The data set that is defined on the OUTDDN DD statement is used instead of the default TRCPUNCH DD statement. For this example, the DD is:

```
//OUTDDN DD ...,DCB=(BLKSIZE=80),...
```

Requests for monitoring and debugging your IMS program

You can use the STAT and LOG requests to help you in debugging your program.

- The Statistics (STAT) call retrieves database statistics.
- The Log (LOG) call makes it possible for the application program to write a record on the system log.

The enhanced OSAM and VSAM STAT calls provide additional information for monitoring performance and fine tuning of the system for specific needs.

When the enhanced STAT call is issued, the following information is returned:

- OSAM statistics for each defined subpool
- VSAM statistics that also include hiperspace statistics
- OSAM and VSAM count fields that have been expanded to 10 digits

Retrieving database statistics: the STAT call

The STAT call is helpful in debugging a program because it retrieves IMS database statistics. It is also helpful in monitoring and fine tuning for performance. The STAT call retrieves OSAM database buffer pool statistics and VSAM database buffer subpool statistics.

This topic contains Product-sensitive Programming Interface information.

When you issue the STAT call, you indicate:

- An I/O area into which the statistics are to be returned.

- A statistics function, which is the name of a 9-byte area whose contents describe the type and format of the statistics you want returned. The contents of the area are defined as follows:
  - The first 4 bytes define the type of statistics desired (OSAM or VSAM).
  - The 5th byte defines the format to be returned (formatted, unformatted, or summary).
  - The remaining 4 bytes are defined as follows:
    - The normal or enhanced STAT call contains 4 bytes of blanks.
    - The extended STAT call contains the 4-byte parameter ' E1 ' (a 1-byte blank, followed by a 2-byte character string, and then another 1-byte blank).

Related reference:

STAT call (Application Programming APIs)

Format of OSAM buffer pool statistics

For OSAM buffer pool statistics, the values are possible for the stat-function parameter and for the format of the data that is returned to the application program. If no OSAM buffer pool is present, a GE status code is returned to the program.

DBASF

This function value provides the full OSAM database buffer pool statistics in a formatted form. The application program I/O area must be at least 360 bytes. Three 120-byte records (formatted for printing) are provided as two heading lines and one line of statistics. The following diagram shows the data format.

```
  BLOCK    FOUND    READS     BUFF     OSAM   BLOCKS     NEW   CHAIN
    REQ  IN POOL   ISSUED     ALTS   WRITES  WRITTEN  BLOCKS  WRITES
nnnnnnn  nnnnnnn    nnnnn  nnnnnnn  nnnnnnn  nnnnnnn   nnnnn   nnnnn

WRITTEN  LOGICAL    PURGE  RELEASE
 AS NEW      CYL      REQ      REQ  ERRORS
          FORMAT
nnnnnnn  nnnnnnn  nnnnnnn  nnnnnnn  nn/nn
```

BLOCK REQ
Number of block requests received.

FOUND IN POOL
Number of times the block requested was found in the buffer pool.

READS ISSUED
Number of OSAM reads issued.

BUFF ALTS
Number of buffers altered in the pool.

OSAM WRITES
Number of OSAM writes issued.

BLOCKS WRITTEN
Number of blocks written from the pool.

NEW BLOCKS
Number of new blocks created in the pool.

CHAIN WRITES
Number of chained OSAM writes issued.

WRITTEN AS NEW
Number of blocks created.

LOGICAL CYL FORMAT
Number of format logical cylinder requests issued.

PURGE REQ
Number of purge user requests.

RELEASE REQ
Number of release ownership requests.

ERRORS
Number of write error buffers currently in the pool or the largest number of errors in the pool during this execution.

DBASU

This function value provides the full OSAM database buffer pool statistics in an unformatted form. The application program I/O area must be at least 72 bytes. Eighteen fullwords of binary data are provided:

Word	Contents
1	A count of the number of words that follow.
2-18	The statistic values in the same sequence as presented by the DBASF function value.

DBASS

This function value provides a summary of the OSAM database buffer pool statistics in a formatted form. The application program I/O area must be at least 180 bytes. Three 60-byte records (formatted for printing) are provided. The following diagram shows the data format.

```
DATA BASE BUFFER POOL:  SIZE nnnnnnn
  REQ1 nnnnn REQ2 nnnnn READ nnnnn WRITES nnnnn LCYL nnnnn
  PURG nnnnn OWNRR nnnnn ERRORS nn/nn
```

SIZE Buffer pool size.

REQ1 Number of block requests.

REQ2 Number of block requests satisfied in the pool plus new blocks created.

READ Number of read requests issued.

WRITES
Number of OSAM writes issued.

LCYL Number of format logical cylinder requests.

PURG Number of purge user requests.

OWNRR
Number of release ownership requests.

ERRORS
Number of permanent errors now in the pool or the largest number of permanent errors during this execution.

Format of VSAM buffer subpool statistics

Because there might be several buffer subpools for VSAM databases, the STAT call is iterative when requesting these statistics. If more than one VSAM local shared

resource pool is defined, statistics are retrieved for all VSAM local shared resource pools in the order in which they are defined. For each local shared resource pool, statistics are retrieved for each subpool according to buffer size.

The first time the call is issued, the statistics for the subpool with the smallest buffer size are provided. For each succeeding call (without intervening use of the PCB), the statistics for the subpool with the next-larger buffer size are provided.

If index subpools exist within the local shared resource pool, the index subpool statistics always follow statistics of the data subpools. Index subpool statistics are also retrieved in ascending order based on the buffer size.

The final call for the series returns a GA status code in the PCB. The statistics returned are totals for all subpools in all local shared resource pools. If no VSAM buffer subpools are present, a GE status code is returned to the program.

VBASF

This function value provides the full VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 360 bytes. Three 120-byte records (formatted for printing) are provided as two heading lines and one line of statistics. Each successive call returns the statistics for the next data subpool. If present, statistics for index subpools follow the statistics for data subpools.

The following diagram shows the data format.

```
                  BUFFER HANDLER STATISTICS
BSIZ NBUF RET RBA RET KEY ISRT ES ISRT KS BFR ALT   BGWRT SYN PTS
nnnK  nnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn

                VSAM STATISTICS POOLID: xxxx
   GETS  SCHBFR   FOUND   READS USR WTS NUR WTS ERRORS
nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn nnnnnnn  nn/nn
```

POOLID
ID of the local shared resource pool.

BSIZ Size of the buffers in this VSAM subpool. In the final call, this field is set to ALL.

NBUF Number of buffers in this subpool. In the final call, this is the number of buffers in all subpools.

RET RBA
Number of retrieve-by-RBA calls received by the buffer handler.

RET KEY
Number of retrieve-by-key calls received by the buffer handler.

ISRT ES
Number of logical records inserted into ESDSs.

ISRT KS
Number of logical records inserted into KSDSs.

BFR ALT
Number of logical records altered in this subpool. Delete calls that result in erasing records from a KSDS are not counted.

BGWRT
Number of times the background-write function was executed by the buffer handler.

SYN PTS
Number of Synchronization calls received by the buffer handler.

GETS Number of VSAM GET calls issued by the buffer handler.

SCHBFR
Number of VSAM SCHBFR calls issued by the buffer handler.

FOUND
Number of times VSAM found the control interval already in the subpool.

READS
Number of times VSAM read a control interval from external storage.

USR WTS
Number of VSAM writes initiated by IMS.

NUR WTS
Number of VSAM writes initiated to make space in the subpool.

ERRORS
Number of write error buffers currently in the subpool or the largest number of write errors in the subpool during this execution.

VBASU

This function value provides the full VSAM database subpool statistics in a unformatted form. The application program I/O area must be at least 72 bytes. Eighteen fullwords of binary data are provided for each subpool:

Word **Contents**

1 A count of the number of words that follow.

2-18 The statistic values in the same sequence as presented by the VBASF function value, except for POOLID, which is not included in this unformatted form.

VBASS

This function value provides a summary of the VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 180 bytes. Three 60-byte records (formatted for printing) are provided.

The following diagram shows the data format.

```
DATA BASE BUFFER POOL:  BSIZE nnnnnnn  POOLID xxxx  Type x
  RRBA nnnnn RKEY nnnnn BFALT nnnnn NREC nnnnn SYN PTS nnnnn
  NMBUFS nnn VRDS nnnnn FOUND nnnnn VWTS nnnnn  ERRORS nn/nn
```

BSIZE Size of the buffers in this VSAM subpool.

POOLID
ID of the local shared resource pool.

TYPE Indicates a data (D) subpool or an index (I) subpool.

RRBA Number of retrieve-by-RBA requests.

RKEY Number of retrieve-by-key requests.

BFALT
Number of logical records altered.

NREC Number of new VSAM logical records created.

SYN PTS
Number of sync point requests.

NMBUFS
Number of buffers in this VSAM subpool.

VRDS Number of VSAM control interval reads.

FOUND
Number of times VSAM found the requested control interval already in the subpool.

VWTS
Number of VSAM control interval writes.

ERRORS
Number of permanent write errors now in the subpool or the largest number of errors in this execution.

Format of enhanced/extended OSAM buffer subpool statistics

The enhanced OSAM buffer pool statistics provide additional information generated for each defined subpool. Because there might be several buffer subpools for OSAM databases, the enhanced `STAT` call repeatedly requests these statistics. The first time the call is issued, the statistics for the subpool with the smallest buffer size is provided. For each succeeding call (without intervening use of the PCB), the statistics for the subpool with the next-larger buffer size is provided.

The final call for the series returns a GA status code in the PCB. The statistics returned are the totals for all subpools. If no OSAM buffer subpools are present, a GE status code is returned.

Extended OSAM buffer pool statistics can be retrieved by including the 4-byte parameter 'bE1b' following the enhanced call function. The extended `STAT` call returns all of the statistics returned with the enhanced call, plus the statistics on the coupling facility buffer invalidates, OSAM caching, and sequential buffering IMMED/SYNC read counts.

Restriction: The extended format parameter is supported by the DBESO, DBESU, and DBESF functions only.

DBESF

This function value provides the full OSAM subpool statistics in a formatted form. The application program I/O area must be at least 600 characters. For OSAM subpools, five 120-byte records (formatted for printing) are provided. Three of the records are heading lines and two of the records are lines of subpool statistics.

The following example shows the enhanced stat call format:

```
      B U F F E R   H A N D L E R   O S A M   S T A T I S T I C S    FIXOPT=X/X   POOLID: xxxx
BSIZ  NBUFS      LOCATE-REQ    NEW-BLOCKS   ALTER- REQ   PURGE- REQ  FND-IN-POOL  BUFRS-SRCH  READ- REQS   BUFSTL-WRT
                   PURGE-WRTS    WT-BUSY-ID   WT-BUSY-WR   WT-BUSY-RD   WT-RLSEOWN   WT-NO-BFRS    ERRORS
nn1K  nnnnnnnn   nnnnnnnnnn    nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn
                   nnnnnnnnnn    nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn   nnnnnnnnnn    nnnnnnn/nnnnnnn
```

The following example shows the extended stat call format:

```
      B U F F E R   H A N D L E R   O S A M   S T A T I S T I C S   STG CLS=     FIXOPT=N/N  POOLID:
BSIZ  NBUFS      LOCATE-REQ    NEW-BLOCKS   ALTER- REQ   PURGE- REQ  FND-IN-POOL  BUFRS-SRCH  READ- REQS   BUFSTL-WRT
                   PURGE-WRTS    WT-BUSY-ID   WT-BUSY-WR   WT-BUSY-RD   WT-RLSEOWN   WT-NO-BFRS    ERRORS
nn1K  nnnnnnn5   nnnnnnnnn0    nnnnnnnnn0   nnnnnnnnn0   nnnnnnnnn0  nnnnnnnnn0   nnnnnnnnn0  nnnnnnnnn0   nnnnnnnnn0
                   nnnnnnnnn    nnnnnnnnn   nnnnnnnnnn   nnnnnnnnn   nnnnnnnnn    nnnnnnnnn    nnnnnn/nnnnnn
      CF-READS   EXPCTD-NF    CFWRT-PRI  CFWRT-CHG  STGCLS-FULL       XI-CNT        VECTR-XI    SB-SEQRD   SB-ANTICIP
      nnnnnnnnn nnnnnnnnn   nnnnnnnnn nnnnnnnnn  nnnnnnnnn   nnnnnnnnn      nnnnnnnnn  nnnnnnnnn   nnnnnnnnn
```

FIXOPT
Fixed options for this subpool. Y or N indicates whether the data buffer prefix and data buffers are fixed.

POOLID
ID of the local shared resource pool.

BSIZ Size of the buffers in this subpool. Set to ALL for total line. For the summary totals (BSIZ=ALL), the FIXOPT and POOLID fields are replaced by an OSM= field. This field is the total size of the OSAM subpool.

NBUFS
Number of buffers in this subpool. This is the total number of buffers in the pool for the ALL line.

LOCATE-REQ
Number of LOCATE-type calls.

NEW-BLOCKS
Number of requests to create new blocks.

ALTER-REQ
Number of buffer alter calls. This count includes NEW BLOCK and BYTALT calls.

PURGE-REQ
Number of PURGE calls.

FND-IN-POOL
Number of LOCATE-type calls for this subpool where data is already in the OSAM pool.

BUFRS-SRCH
Number of buffers searched by all LOCATE-type calls.

READ-REQS
Number of READ I/O requests.

BUFSTL-WRT
Number of single block writes initiated by buffer steal routine.

PURGE-WRTS
Number of blocks for this subpool written by purge.

WT-BUSY-ID
Number of LOCATE calls that waited due to busy ID.

WT-BUSY-WR
Number of LOCATE calls that waited due to buffer busy writing.

WT-BUSY-RD
Number of LOCATE calls that waited due to buffer busy reading.

WT-RLSEOWN
Number of buffer steal or purge requests that waited for ownership to be released.

WT-NO-BFRS
Number of buffer steal requests that waited because no buffers are available to be stolen.

ERRORS
Total number of I/O errors for this subpool or the number of buffers locked in pool due to write errors.

CF-READS
Number of blocks read from CF.

EXPCTD-NF
Number of blocks expected but not read.

CFWRT-PRI
Number of blocks written to CF (prime).

CFWRT-CHG
Number of blocks written to CF (changed).

STGGLS-FULL
Number of blocks not written (STG CLS full).

XI-CNTL
Number of XI buffer invalidate calls.

VECTR-XI
Number of buffers found invalidated by XI on VECTOR call.

SB-SEQRD
Number of immediate (SYNC) sequential reads (SB stat).

SB-ANTICIP
Number of anticipatory reads (SB stat).

DBESU

This function value provides full OSAM statistics in an unformatted form. The application program I/O area must be at least 84 bytes. Twenty-one fullwords of binary data are provided for each subpool:

Word	Contents
1	A count of the number of words that follow.
2-19	The statistics provided in the same sequence as presented by the DBESF function value.
20	The POOLID provided at subpool definition time.
21	The second byte contains the following fix options for this subpool: • X'04' = DATA BUFFER PREFIX fixed • X'02' = DATA BUFFERS fixed The summary totals (word 2=ALL), for word 21, contain the total size of the OSAM pool.
22-30	Extended stat data in same sequence as on DBESF call.

DBESS

This function value provides a summary of the OSAM database buffer pool statistics in a formatted form. The application program I/O area must be at least 360 bytes. Six 60-byte records (formatted for printing) are provided. This `STAT` call is a restructured DBASF `STAT` call that allows for 10-digit count fields. In addition, the subpool header blocks give a total of the number of OSAM buffers in the pool.

The following shows the data format:

```
DATA BASE BUFFER POOL:  NSUBPL nnnnnn  NBUFS nnnnnnnn
  BLKREQ nnnnnnnnnn   INPOOL nnnnnnnnnn   READS  nnnnnnnnnn
  BUFALT nnnnnnnnnn   WRITES nnnnnnnnnn   BLKWRT nnnnnnnnnn
  NEWBLK nnnnnnnnnn   CHNWRT nnnnnnnnnn   WRTNEW nnnnnnnnnn
  LCYLFM nnnnnnnnnn   PURGRQ nnnnnnnnnn   RLSERQ nnnnnnnnnn
  FRCWRT nnnnnnnnnn   ERRORS nnnnnnnn/nnnnnnnn
```

NSUBPL
Number of subpools defined for the OSAM buffer pool.

NBUFS
Total number of buffers defined in the OSAM buffer pool.

BLKREQ
Number of block requests received.

INPOOL
Number of times the block requested is found in the buffer pool.

READS
Number of OSAM reads issued.

BUFALT
Number of buffers altered in the pool.

WRITES
Number of OSAM writes issued.

BLKWRT
Number of blocks written from the pool.

NEWBLK
Number of blocks created in the pool.

CHNWRT
Number of chained OSAM writes issued.

WRTNEW
Number of blocks created.

LCYLFM
Number of format logical cylinder requests issued.

PURGRQ
Number of purge user requests.

RLSERQ
Number of release ownership requests.

FRCWRT
Number of forced write calls.

ERRORS
Number of write error buffers currently in the pool or the largest number of errors in the pool during this execution.

DBESO

This function value provides the full OSAM database subpool statistics in a formatted form for online statistics that are returned as a result of a `/DIS POOL` command. This call can also be a user-application `STAT` call. When issued as an application DL/I `STAT` call, the program I/O area must be at least 360 bytes. Six 60-byte records (formatted for printing) are provided.

Example: The following shows the enhanced stat call format:

```
OSAM DB BUFFER POOL:ID xxxx BSIZE nnnnnK NBUFnnnnnnn  FX=X/X
  LCTREQ  nnnnnnnnnn  NEWBLK  nnnnnnnnnn  ALTREQ  nnnnnnnnnn
  PURGRQ  nnnnnnnnnn  FNDIPL  nnnnnnnnnn  BFSRCH  nnnnnnnnnn
  RDREQ   nnnnnnnnnn  BFSTLW  nnnnnnnnnn  PURGWR  nnnnnnnnnn
  WBSYID  nnnnnnnnnn  WBSYWR  nnnnnnnnnn  WBSYRD  nnnnnnnnnn
  WRLSEO  nnnnnnnnnn  WNOBFR  nnnnnnnnnn  ERRORS nnnnn/nnnnn
```

Example: The following shows the extended stat call format:

```
OSAM DB BUFFER POOL:ID xxxx BSIZE nnnnnK NBUFnnnnnnn  FX=X/X
  LCTREQ  nnnnnnnnnn  NEWBLK  nnnnnnnnnn  ALTREQ  nnnnnnnnnn
  PURGRQ  nnnnnnnnnn  FNDIPL  nnnnnnnnnn  BFSRCH  nnnnnnnnnn
  RDREQ   nnnnnnnnnn  BFSTLW  nnnnnnnnnn  PURGWR  nnnnnnnnnn
  WBSYID  nnnnnnnnnn  WBSYWR  nnnnnnnnnn  WBSYRD  nnnnnnnnnn
  WRLSEO  nnnnnnnnnn  WNOBFR  nnnnnnnnnn  ERRORS nnnnn/nnnnn
  CFREAD  nnnnnnnnnn  CFEXPC  nnnnnnnnnn  CFWRPR nnnnn/nnnnn
  CFWRCH  nnnnnnnnnn  STGCLF  nnnnnnnnnn  XIINV  nnnnn/nnnnn
  XICLCT  nnnnnnnnnn  SBSEQR  nnnnnnnnnn  SBANTR nnnnn/nnnnn
```

POOLID
ID of the local shared resource pool.

BSIZE Size of the buffers in this subpool. Set to ALL for summary total line. For the summary totals (BSIZE=ALL), the FX= field is replaced by the OSAM= field. This field is the total size of the OSAM buffer pool. The POOLID is not shown. For the summary totals (BSIZE=ALL), the FX= field is replaced by the OSAM= field. This field is the total size of the OSAM buffer pool. The POOLID is not shown.

NBUF Number of buffers in this subpool. Total number of buffers in the pool for the ALL line.

FX= Fixed options for this subpool. Y or N indicates whether the data buffer prefix and data buffers are fixed.

LCTREQ
Number of LOCATE-type calls.

NEWBLK
Number of requests to create new blocks.

ALTREQ
Number of buffer alter calls. This count includes NEW BLOCK and BYTALT calls.

PURGRQ
Number of PURGE calls.

FNDIPL
Number of LOCATE-type calls for this subpool where data is already in the OSAM pool.

BFSRCH
Number of buffers searched by all LOCATE-type calls.

RDREQ
Number of READ I/O requests.

BFSTLW
Number of single-block writes initiated by buffer-steal routine.

PURGWR
Number of buffers written by purge.

WBSYID
Number of LOCATE calls that waited due to busy ID.

WBSYWR
Number of LOCATE calls that waited due to buffer busy writing.

WBSYRD
Number of LOCATE calls that waited due to buffer busy reading.

WRLSEO
Number of buffer steal or purge requests that waited for ownership to be released.

WNOBRF
Number of buffer steal requests that waited because no buffers are available to be stolen.

ERRORS
Total number of I/O errors for this subpool or the number of buffers locked in pool due to write errors.

CFREAD
Number of blocks read from CF.

CFEXPC
Number of blocks expected but not read.

CFWRPR
Number of blocks written to CF (prime).

CFWRCH
Number of blocks written to CF (changed).

STGCLF
Number of blocks not written (STG CLS full).

XIINV
Number of XI buffer invalidate calls.

XICLCT
Number of buffers found invalidated by XI on VECTOR call.

SBSEQR
Number of immediate (SYNC) sequential reads (SB stat).

SBANTR
Number of anticipatory reads (SB stat).

Format of enhanced VSAM buffer subpool statistics

The enhanced VSAM buffer subpool statistics provide information on the total size of VSAM subpools in virtual storage and in hiperspace. All count fields are 10 digits.

Because there might be several buffer subpools for VSAM databases, the enhanced `STAT` call repeatedly requests these statistics. If more than one VSAM local shared resource pool is defined, statistics are retrieved for all VSAM local shared resource pools in the order in which they are defined. For each local shared resource pool, statistics are retrieved for each subpool according to buffer size.

The first time the call is issued, the statistics for the subpool with the smallest buffer size are provided. For each succeeding call (without intervening use of the PCB), the statistics for the subpool with the next-larger buffer size are provided.

If index subpools exist within the local shared resource pool, the index subpool statistics always follow the data subpools statistics. Index subpool statistics are also retrieved in ascending order based on the buffer size.

The final call for the series returns a GA status code in the PCB. The statistics returned are totals for all subpools in all local shared resource pools. If no VSAM buffer subpools are present, a GE status code is returned to the program.

VBESF

This function value provides the full VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 600 bytes. For each shared resource pool ID, the first call returns five 120-byte records (formatted for printing). Three of the records are heading lines and two of the records are lines of subpool statistics.

The following shows the data format:

```
     B U F F E R   H A N D L E R   S T A T I S T I C S  /  V S A M   S T A T I S T I C S   FIXOPT=X/X/X   POOLID: xxxx
BSIZ NBUFFRS HS-NBUF  RETURN-RBA  RETURN-KEY  ESDS-INSRT  KSDS-INSRT  BUFFRS-ALT  BKGRND-WRT  SYNC-POINT  ERRORS
             VSAM-GETS  SCHED-BUFR  VSAM-FOUND  VSAM-READS  USER-WRITS   VSAM-WRITS  HSRDS-SUCC  HSWRT-SUCC HSR/W-FAIL
nn1K  nnnnnn nnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnn/nnnnnn
              nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn   nnnnnnnnnn  nnnnnnnnnn  nnnnnnnnnn nnnnn/nnnnn
```

FIXOPT
Fixed options for this subpool. Y or N indicates whether the data buffer prefix, the index buffers, and the data buffers are fixed.

POOLID
ID of the local shared resource pool.

BSIZ Size of the buffers in this subpool. Set to ALL for total line. For the summary totals (BSIZ=ALL), the FIXOPT and POOLID fields are replaced by a VS= field and a HS= field. The VS= field is the total size of the VSAM subpool in virtual storage. The HS= field is the total size of the VSAM subpool in hiperspace.

NBUFFRS
Number of buffers in this subpool. Total number of buffers in the VSAM pool that appears in the ALL line.

HS-NBUF
Number of hiperspace buffers defined for this subpool.

RETURN-RBA
Number of retrieve-by-RBA calls received by the buffer handler.

RETURN-KEY
Number of retrieve-by-key calls received by the buffer handler.

ESDS-INSRT
Number of logical records inserted into ESDSs.

KSDS-INSRT
Number of logical records inserted into KSDSs.

BUFFRS-ALT
Number of logical records altered in this subpool. Delete calls that result in erasing records from a KSDS are not counted.

BKGRND-WRT
Number of times the background write function was executed by the buffer handler.

SYNC-POINT
Number of Synchronization calls received by the buffer handler.

ERRORS
Number of write error buffers currently in the subpool or the largest number of write errors in the subpool during this execution.

VSAM-GETS
Number of VSAM Get calls issued by the buffer handler.

SCHED-BUFR
Number of VSAM Scheduled-Buffer calls issued by the buffer handler

VSAM-FOUND
Number of times VSAM found the control interval in the buffer pool.

VSAM-READS
Number of times VSAM read a control interval from external storage.

USER-WRITS
Number of VSAM writes initiated by IMS.

VSAM-WRITS
Number of VSAM writes initiated to make space in the subpool.

HSRDS-SUCC
Number of successful VSAM reads from hiperspace buffers.

HSWRT-SUCC
Number of successful VSAM writes from hiperspace buffers.

HSR/W-FAIL
Number of failed VSAM reads from hiperspace buffers/number of failed VSAM writes to hiperspace buffers. This indicates the number of times a VSAM READ/WRITE request from or to hiperspace resulted in DASD I/O.

VBESU

This function value provides full VSAM statistics in an unformatted form. The application program I/O area must be at least 104 bytes. Twenty-five fullwords of binary data are provided for each subpool.

Word	Contents
1	A count of the number of words that follow.
2-23	The statistics provided in the same sequence as presented by the VBESF function value.
24	The POOLID provided at the time the subpool is defined.
25	The first byte contains the subpool type, and the third byte contains the following fixed options for this subpool: • X'08' = INDEX BUFFERS fixed • X'04' = DATA BUFFER PREFIX fixed • X'02' = DATA BUFFERS fixed The summary totals (word 2=ALL) for word 25 and word 26 contain the virtual and hiperspace pool sizes.

VBESS

This function value provides a summary of the VSAM database subpool statistics in a formatted form. The application program I/O area must be at least 360 bytes. For each shared resource pool ID, the first call provides six 60-byte records (formatted for printing).

The following shows the data format:

```
VSAM DB BUFFER POOL:ID xxxx   BSIZE nnnnnnK   TYPE x   FX=X/X/X
  RRBA  nnnnnnnnnn    RKEY       nnnnnnnnnn   BFALT  nnnnnnnnnn
  NREC  nnnnnnnnnn    SYNC PT    nnnnnnnnnn   NBUFS  nnnnnnnnnn
  VRDS  nnnnnnnnnn    FOUND      nnnnnnnnnn   VWTS   nnnnnnnnnn
  HSR-S nnnnnnnnnn    HSW-S      nnnnnnnnnn   HS NBUFS nnnnnnnn
  HS-R/W-FAIL   nnnnn/nnnnn       ERRORS    nnnnnn/nnnnnn
```

POOLID
ID of the local shared resource pool.

BSIZE Size of the buffers in this VSAM subpool.

TYPE Indicates a data (D) subpool or an index (I) subpool.

FX Fixed options for this subpool. Y or N indicates whether the data buffer prefix, the index buffers, and the data buffers are fixed.

RRBA
Number of retrieve-by-RBA calls received by the buffer handler.

RKEY Number of retrieve-by-key calls received by the buffer handler.

BFALT
Number of logical records altered.

NREC Number of new VSAM logical records created.

SYNC PT
Number of sync point requests.

NBUFS
Number of buffers in this VSAM subpool.

VRDS Number of VSAM control interval reads.

FOUND
Number of times VSAM found the requested control interval already in the subpool.

VWTS
Number of VSAM control interval writes.

HSR-S
Number of successful VSAM reads from hiperspace buffers.

HSW-S
Number of successful VSAM writes to hiperspace buffers.

HS NBUFS
Number of VSAM hiperspace buffers defined for this subpool.

HS-R/W-FAIL
Number of failed VSAM reads from hiperspace buffers and number of failed VSAM writes to hiperspace buffers. This indicates the number of times a VSAM READ/WRITE request to or from hiperspace resulted in DASD I/O.

ERRORS
Number of permanent write errors now in the subpool or the largest number of errors in this execution.

Writing Information to the system log: the LOG request

An application program can write a record to the system log by issuing the LOG call.

When you issue the LOG request, you specify the I/O area that contains the record you want written to the system log. You can write any information to the log that you want, and you can use different log codes to distinguish between different types of information.

Related Reading: For information about coding the LOG request, see the appropriate application programming reference information.

What to do when your IMS program terminates abnormally

When your program terminates abnormally, you can take the following actions to simplify the task of finding and fixing the problem.

- Record as much information as possible about the circumstances under which the program terminated abnormally.
- Check for certain initialization and execution errors.

Recommended actions after an abnormal termination of an IMS program

Many places have guidelines on what you should do if your program terminates abnormally. The suggestions given here are common guidelines:

- Document the error situation to help in investigating and correcting it. The following information can be helpful:
  - The program's PSB name
  - The transaction code that the program was processing (online programs only)
  - The text of the input message being processed (online programs only)
  - The call function
  - The name of the originating logical terminal (online programs only)
  - The contents of the PCB that was referenced in the call that was executing
  - The contents of the I/O area when the problem occurred
  - If a database call was executing, the SSAs, if any, that the call used
  - The date and time of day
- When your program encounters an error, it can pass all the required error information to a standard error routine. You should not use STAE or ESTAE routines in your program; IMS uses STAE or ESTAE routines to notify the control region of any abnormal termination of the application program. If you call your own STAE or ESTAE routines, IMS may not get control if an abnormal termination occurs.
- Online programs might want to send a message to the originating logical terminal to inform the person at the terminal that an error has occurred. Unless you are using a CCTL, your program can get the logical terminal name from the I/O PCB, place it in an express PCB, and issue one or more ISRT calls to send the message.

- An online program might also want to send a message to the master terminal operator giving information about the program's termination. To do this, the program places the logical terminal name of the master terminal in an express PCB and issues one or more ISRT calls. (This is not applicable if you are using a CCTL.)
- You might also want to send a message to a printer so that you will have a hard-copy record of the error.
- You can send a message to the system log by issuing a LOG request.
- Some places run a BMP at the end of the day to list all the errors that have occurred during the day. If your shop does this, you can send a message using an express PCB that has its destination set for that BMP. (This is not applicable if you are using a CCTL.)

Diagnosing an abnormal termination of an IMS program

If your program does not run correctly when you are testing it or when it is executing, you need to isolate the problem. The problem might be anything from a programming error (for example, an error in the way you coded one of your requests) to a system problem. This section gives some guidelines about the steps that you, as the application programmer, can take when your program fails to run, terminates abnormally, or gives incorrect results.

IMS program initialization errors

Before your program receives control, IMS must have correctly loaded and initialized the PSB and DBDs used by your application program. Often, when the problem is in this area, you need a system programmer or DBA (or your equivalent specialist) to fix the problem. One thing you can do is to find out if there have been any recent changes to the DBDs, PSB, and the control blocks that they generate.

IMS program execution errors

If you do not have any initialization errors, check:

1. The output from the compiler. Make sure that all error messages have been resolved.
2. The output from the binder:
   - Are all external references resolved?
   - Have all necessary modules been included?
   - Was the language interface module correctly included?
   - Is the correct entry point specified?
3. Your JCL:
   - Is the information that described the files that contain the databases correct? If not, check with your DBA.
   - Have you included the DL/I parameter statement in the correct format?
   - Have you included the region size parameter in the EXEC statement? Does it specify a region or partition large enough for the storage required for IMS and your program?
   - Have you declared the fields in the PCB masks correctly?

- If your program is an assembler language program, have you saved and restored registers correctly? Did you save the list of PCB addresses at entry? Does register 1 point to a parameter list of fullwords before issuing any DL/I calls?
- For COBOL for z/OS and PL/I for MVS™ and VM, are the literals you are using for arguments in DL/I calls producing the results you expect? For example, in PL/I for MVS and VM, is the parameter count being generated as a half-word instead of a fullword, and is the function code producing the required 4-byte field?
- Use the PCB as much as possible to determine what in your program is producing incorrect results.

Related concepts:

"Use of STAE or ESTAE and SPIE in IMS programs" on page 55

Chapter 9. Testing a CICS application program

You should perform a program unit test on your CICS application program to ensure that the program correctly handles its input data, processing, and output data. The amount and type of testing you do depends on the individual program.

Recommendations for testing a CICS program

When you are ready to test your program, be aware of your established test procedures before you start.

To start testing, you need the following three items:

- Test JCL.
- A test database. When you are testing a program, do not execute it against a production database because the program, if faulty, might damage valid data.
- Test input data. The input data that you use need not be current, but it should be valid data. You cannot be sure that your output data is valid unless you use valid input data.

The purpose of testing the program is to make sure that the program can correctly handle all the situations that it might encounter.

To thoroughly test the program, try to test as many of the paths that the program can take as possible. For example:

- Test each path in the program by using input data that forces the program to execute each of its branches.
- Be sure that your program tests its error routines. Again, use input data that will force the program to test as many error conditions as possible.
- Test the editing routines your program uses. Give the program as many different data combinations as possible to make sure it correctly edits its input data.

Testing your CICS program

You can use different tools to test a CICS program, depending on the type of program.

The following table summarizes the tools that are available for online DBCTL, batch, and BMP programs.

Table 30. Tools you can use for testing your program.

Tool	Online (DBCTL)	Batch	BMP
Execution Diagnostic Facility (EDF)	Yes[1]	No	No
CICS dump control	Yes	No	No
CICS trace control	Yes	Yes	No
DFSDDLT0	No	Yes[2]	Yes[2]
DL/I image capture program	Yes	Yes	Yes

© Copyright IBM Corp. 1974, 2015

Table 30. Tools you can use for testing your program (continued).

Tool	Online (DBCTL)	Batch	BMP

Notes:

1. For online, command-level programs only.
2. For call-level programs only. (For a command-level batch program, you can use DL/I image capture program first, to produce calls for DFSDDLT0.)

Using the Execution Diagnostic Facility (command-level only)

You can use the Execution Diagnostic Facility (EDF) to test command-level programs online. EDF can display EXEC CICS and EXEC DLI commands in online programs; it cannot intercept DL/I calls.

With EDF you can:

- Display and modify working storage; you can change values in the DIB.
- Display and modify a command before it is executed. You can modify the value of any argument, and then execute the command.
- Modify the return codes after the execution of the command. After the command has been executed, but before control is returned to the application program, the command is intercepted to show the response and any argument values set by CICS.

You can run EDF on the same terminal as the program you are testing.

Related Reading: For more information about using EDF, see "Execution (Command-Level) Diagnostic Facility" in CICS Transaction Server for z/OS CICS Application Programming Reference.

Using CICS dump control

You can use the CICS dump control facility to dump virtual storage areas, CICS tables, and task-related storage areas. For more information about using the CICS dump control facility, see the CICS application programming reference manual that applies to your version of CICS.

Using CICS trace control

You can use the trace control facility to help debug and monitor your online programs in the DBCTL environment. You can use trace control requests to record entries in a trace table. The trace table can be located either in virtual storage or on auxiliary storage. If it is in virtual storage, you can gain access to it by investigating a dump; if it is on auxiliary storage, you can print the trace table. For more information about the control statements you can use to produce trace entries, see the information about trace control in the application programming reference manual that applies to your version of CICS.

Tracing DL/I calls with image capture

DL/I image capture program (DFSDLTR0) is a trace program that can trace and record DL/I calls issued by batch, BMP, and online (DBCTL environment) programs. You can also use the image capture program with command-level programs, and you can produce calls for use as input to DFSDDLT0.

You can use the image capture program to:

- **Test your program**

  If the image capture program detects an error in a call it traces, it reproduces as much of the call as possible, although it cannot document where the error occurred, and cannot always reproduce the full SSA.

- **Produce input for DFSDDLT0 (DL/I test program)**

  You can use the output produced by the image capture program as input to DFSDDLT0. The image capture program produces status statements, comment statements, call statements, and compare statements for DFSDDLT0. For example, you can use the image capture program with a command-level program, to produce calls for DFSDDLT0.

- **Debug your program**

  When your program terminates abnormally, you can rerun the program using the image capture program. The image capture program can then reproduce and document the conditions that led to the program failure. You can use the information in the report produced by the image capture program to find and fix the problem.

Using image capture with DFSDDLT0

The image capture program produces the following control statements that you can use as input to DFSDDLT0:

- **Status statements**

  When you invoke the image capture program, it produces the status statement. The status statement it produces:

  – Sets print options so that DFSDDLT0 prints all call trace comments, all DL/I calls, and the results of all comparisons.

  – Determines the new relative PCB number each time a PCB change occurs while the application program is executing.

- **Comments statement**

  The image capture program also produces a comments statement when you invoke it. The comments statements give:

  – The time and date IMS started the trace

  – The name of the PSB being traced

  The image capture program also produces a comments statement preceding any call in which IMS finds an error.

- **Call statements**

  The image capture program produces a call statement for each DL/I call or EXEC DLI command the application program issues. It also generates a `CHKP` call when it starts the trace and after each commit point or `CHKP` request.

- **Compare statements**

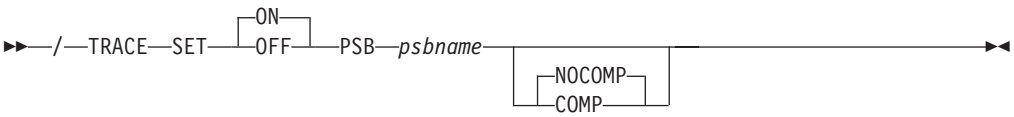
  If you specify COMP on the DLITRACE control statement, the image capture program produces data and PCB comparison statements.

Running image capture online

When you run the image capture program online, the trace output goes to the IMS log data set. To run the image capture program online, you issue the IMS `TRACE` command from the z/OS console.

If you trace a BMP and you want to use the trace results with DFSDDLT0, the BMP must have exclusive write access to the databases it processes. If the application program does not have exclusive access, the results of DFSDDLT0 may differ from the results of the application program.

The following diagram shows TRACE command format:



SET ON|OFF
Turns the trace on or off.

PSB psbname
Specifies the name of the PSB you want to trace. You can trace more than one PSB at the same time, by issuing a separate TRACE command for each PSB.

COMP|NOCOMP
Specifies whether you want the image capture program to produce data and PCB compare statements to be used with DFSDDLT0.

Running image capture as a batch job

To run the image capture program as a batch job, you use the DLITRACE control statement in the DFSVSAMP DD data set.

In the DLITRACE control statement, you specify:

- Whether you want to trace all of the DL/I calls the program issues or trace only a certain group of calls.
- Whether you want the trace output to go to:
  - A sequential data set that you specify
  - The IMS log data set
  - Both sequential and IMS log data sets

If the program being traced issues CHKP and XRST calls, the checkpoint and restart information may not be directly reproducible when you use the trace output with DFSDDLT0.

When you run DFSDDLT0 in an IMS DL/I or DBB batch region with trace output, the results are the same as the application program's results, but only if the database has not been altered.

For information on the format of the DLITRACE control statement in the DFSVSAMP DD data set, see the topic "Defining DL/I call image trace" in *IMS Version 11 System Definition*.

Example of DLITRACE

This example shows a DLITRACE control statement that traces the first 14 DL/I calls or commands that the program issues, sends the output to the IMS log data set, and produces data and PCB comparison statements for DFSDDLT0.

```
//DFSVSAMP DD *
DLITRACE LOG=YES,STOP=14,COMP
/*
```

Special JCL requirements

The following are special JCL requirements:

//IEFRDER DD
If you want log data set output, this DD statement is required to define the IMS log data set.

//DFSTROUT DD|anyname
If you want sequential data set output, this DD statement is required to define that data set. If you want to specify an alternate DDNAME (anyname), it must be specified using the DDNAME parameter on the DLITRACE control statement.

The DCB parameters on the JCL statement are not required. The data set characteristics are:

- RECFM=F
- LRECL=80

Notes on using image capture

- If the program being traced issues CHKP and XRST calls, the checkpoint and restart information may not be directly reproducible when you use the trace output with the DFSDDLT0.
- When you run DFSDDLT0 in an IMS DL/I or DBB batch region with trace output, the results are the same as the application program's results provided the database has not been altered.

Retrieving image capture data from the log data set

If the trace output is sent to the IMS log data set, you can retrieve it by using utility DFSERA10 and a DL/I call trace exit routine, DFSERA50. DFSERA50 deblocks, formats, and numbers the image capture program records to be retrieved. To use DFSERA50, you must insert a DD statement defining a sequential output data set in the DFSERA10 input stream. The default ddname for this DD statement is TRCPUNCH. The card must specify BLKSIZE=80.

For example, you can use the following examples of DFSERA10 input control statements in the SYSIN data set to retrieve the image capture program data from the log data set:

- **Print all image capture program records:**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,FLDTYP=X
```

- **Print selected image capture program records by PSB name:**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT OFFSET=25,VLDTYP=C,FLDLEN=8,
               VALUE=psbname, COND=E
```

- **Format image capture program records (in a format that can be used as input to DFSDDLT0):**

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT EXITR=DFSERA50,OFFSET=25,FLDTYP=C
               VALUE=psbname,FLDLEN=8,DDNAME=OUTDDN,COND=E
```

The DDNAME= parameter is used to name the DD statement used by DFSERA50. The data set defined on the OUTDDN DD statement is used instead of the default TRCPUNCH DD statement. For this example, the DD appears as:

```
//OUTDDN DD ...,DCB=(BLKSIZE=80),...
```

Requests for monitoring and debugging your CICS program

You can use the STAT and LOG requests to help you in debugging your program.

- The statistics (STAT) request retrieves database statistics. STAT can be issued from both call- and command-level programs.
- The log (LOG) request makes it possible for the application program to write a record on the system log. You can issue LOG as a command or call in a batch program; in this case, the record is written to the IMS log. You can issue LOG as a call or command in an online program in the DBCTL environment; in this case, the record is written to the DBCTL log.

What to do when your CICS program terminates abnormally

Whenever your program terminates abnormally, you can take some actions to simplify the task of finding and fixing the problem.

First, you can record as much information as possible about the circumstances under which the program terminated abnormally; and second, you can check for certain initialization and execution errors.

Recommended actions after an abnormal termination of CICS

Many places have guidelines on what you should do if your program terminates abnormally. The suggestions given here are some common guidelines:

- Document the error situation to help in investigating and correcting it. Some of the information that can be helpful is:
  - The program's PSB name
  - The transaction code that the program was processing (online programs only)
  - The text of the input screen being processed (online programs only)
  - The call function
  - The terminal ID (online programs only)
  - The contents of the PCB or the DIB
  - The contents of the I/O area when the problem occurred
  - If a database request was executing, the SSAs or SEGMENT and WHERE options, if any, the request used
  - The date and time of day
- When your program encounters an error, it can pass all the required error information to a standard error routine.
- An online program might also want to send a message to the master terminal destination (CSMT) and application terminal operator, giving information about the program's termination.
- You can send a message to the system log by issuing a LOG request.

Diagnosing an abnormal termination of CICS

If your program does not run correctly when you are testing it or when it is executing, you need to isolate the problem. The problem might be anything from a

programming error (for example, an error in the way you coded one of your requests) to a system problem. This section gives some guidelines about the steps that you, as the application programmer, can take when your program fails to run, terminates abnormally, or gives incorrect results.

CICS initialization errors

Before your program receives control, IMS must have correctly loaded and initialized the PSB and DBDs used by your application program. Often, when the problem is in this area, you need a system programmer or DBA (or your equivalent specialist) to fix the problem. One thing you can do is to find out if there have been any recent changes to the DBDs, PSB, and the control blocks that they generate.

CICS execution errors

If you do not have any initialization errors, check the following in your program:

1. The output from the compiler. Make sure that all error messages have been resolved.
2. The output from the binder:
   - Are all external references resolved?
   - Have all necessary modules been included?
   - Was the language interface module correctly included?
   - Is the correct entry point specified (for batch programs only)?
3. Your JCL:
   - Is the information that described the files that contain the databases correct? If not, check with your DBA.
   - Have you included the DL/I parameter statement in the correct format (for batch programs only)?
   - Have you included the region size parameter in the EXEC statement? Does it specify a region or partition large enough for the storage required for IMS and your program (for batch programs only)?
4. Your call-level program:
   - Have you declared the fields in the PCB masks correctly?
   - If your program is an assembler language program, have you saved and restored registers correctly? Did you save the list of PCB addresses at entry? Does register 1 point to a parameter list of full words before issuing any DL/I calls?
   - For COBOL for z/OS and PL/I for MVS and VM, are the literals you are using for arguments in DL/I calls producing the results you expect? For example, in PL/I for MVS and VM, is the parameter count being generated as a half word instead of a fullword, and is the function code producing the required 4-byte field?
   - Use the PCB as much as possible to determine what in your program is producing incorrect results.
5. Your command-level program:
   - Did you use the `FROM` option with your `ISRT` or `REPL` command? If not, data will not be transferred to the database.
   - Check translator messages for errors.

Chapter 10. Documenting your application program

Many places establish standards for program documentation; make sure you are aware of your established standards.

Documentation for other programmers

Documenting a program is not something you do at the end of the project; your documentation will be much more complete, and more useful to others, if you record information about the program as you structure and code it. Include any information that might be useful to someone else who must work with your program.

The reason you record this information is so that people who maintain your program know why you chose certain commands, options, call structures, and command codes. For example, if the DBA were considering reorganizing the database in some way, information about why your program accesses the data the way it does would be helpful.

Information you can include for other programmers includes:

- Flowcharts and pseudocode for the program
- Comments about the program from code inspections
- A written description of the program flow
- Information about why you chose the call sequence you did, such as:
  - Did you test the call sequence using DFSDDLT0?
  - In cases where more than one combination of calls would have had the same results, why did you choose the sequence you did?
  - What was the other sequence? Did you test it using DFSDDLT0?
- Any problems you encountered in structuring or coding the program
- Any problems you had when you tested the program
- Warnings about what should not be changed in the program

All this information relates to structuring and coding the program. In addition, you should include the documentation for end users with the documentation for programmers.

Ultimately, you must determine the level of detail necessary and the most suitable form for documenting the program. These documentation guidelines are provided as suggestions.

Documentation for end users

In addition to documenting the design of the application, you should record information about how the program is used.

The amount of information that users need and how much of it you should supply depends upon whom the users of the program are and what type of program it is.

At a minimum, include the following information for those who use your program:

- What one needs in order to use the program, for example:

© Copyright IBM Corp. 1974, 2015

  - For online programs, is there a password?
  - For batch programs, what is the required JCL?
- The input that one needs to supply to the program, for example:
  - For an MPP, what is the MOD name that must be entered to initially format the screen?
  - For a CICS online program, what is the CICS transaction code that must be entered? What terminal input is expected?
  - For a batch program, is the input in the form of a tape, or a disk data set? Is the input originally output from a previous job?
- The content and form of the program's output, for example:
  - If it is a report, show the format or include a sample listing.
  - For an online application program, show what the screen will look like.
- For online programs, if decisions must be made, explain what is involved in each decision. Present the choices and the defaults.

If the people that will be using your program are unfamiliar with terminals, they will need a user's guide also. This guide should give explicit instructions on how to use the terminal and what a user can expect from the program. The guide should contain discussions of what should be done if the task or program abends, whether the program should be restarted, or if the database requires recovery. Although you may not be responsible for providing this kind of information, you should provide any information that is unique to your application to whomever is responsible for this kind of information.

Part 2. Application programming for IMS DB

IMS provides support for writing application programs to access the IMS database.

© Copyright IBM Corp. 1974, 2015

Chapter 11. Writing your application programs for IMS DB

You can write application programs in High Level Assembler language, C language, COBOL, Java, Pascal, and PL/I to access data in the IMS DB.

Related concepts:

Chapter 35, "IMS solutions for Java development overview," on page 539

Programming guidelines

The number, type, and sequence of the IMS requests your program issues affects the efficiency of your program. A program that is poorly designed can still run if it is coded correctly. IMS will not find design errors for you. The suggestions that follow will help you develop the most efficient design possible for your application program.

When you have a general sequence of calls mapped out for your program, look over the guidelines on sequence to see if you can improve it. An efficient sequence of requests results in efficient internal IMS processing. As you write your program, keep in mind the guidelines explained in this section. The following list offers programming guidelines that will help you write efficient and error-free programs.

- Use the most simple call. Qualify your requests to narrow the search for IMS.
- Use the request or sequence of requests that will give IMS the shortest path to the segment you want.
- Use as few requests as possible. Each DL/I call your program issues uses system time and resources. You may be able to eliminate unnecessary calls by:
  - Using path requests when you are replacing, retrieving, or inserting more than one segment in the same path. If you are using more than one request to do this, you are issuing unnecessary requests.
  - Changing the sequence so that your program saves the segment in a separate I/O area, and then gets it from that I/O area the subsequent times it needs the segment. If your program retrieves the same segment more than once during program execution, you are issuing unnecessary requests.
  - Anticipating and eliminating needless and nonproductive requests, such as requests that result in GB, GE, and II status codes. For example, if you are issuing `GN` calls for a particular segment type, and you know how many occurrences of that segment type exist, do not issue the `GN` that results in a GE status code. Keep track of the number of occurrences your program retrieves, and then continue with other processing when you know you have retrieved all the occurrences of that segment type.
  - Issuing an insert request with a qualification for each parent, rather than issuing Get requests for the parents to make sure that they exist. If IMS returns a GE status code, at least one of the parents does not exist. When you are inserting segments, you cannot insert dependent segments unless the parent segments exist.
- Commit your updates regularly. IMS limits full-function databases so that only 300 databases at a time can have uncommitted updates. Logically related databases, secondary indexes, and HALDB partitions are counted towards this limit. The number of partitions in HALDB databases is the most common reason for approaching the 300 database limit for uncommitted updates. If the PROCOPT values allow a BMP application to insert, replace, or delete segments

© Copyright IBM Corp. 1974, 2015

in the databases, ensure that the BMP application does not update a combined total of more than 300 databases and HALDB partitions without committing the changes.

- Keep the main section of the program logic together. For example, branch to conditional routines, such as error and print routines in other parts of the program, instead of branching around them to continue normal processing.
- Use call sequences that make good use of the physical placement of the data. Access segments in hierarchic sequence as often as possible, and avoid moving backward in the hierarchy.
- Process database records in order of the key field of the root segments. (For HDAM and PHDAM databases, this order depends on the randomizing routine that is used. Check with your DBA for this information.)
- Avoid constructing the logic of the program and the structure of commands or calls in a way that depends heavily on the database structure. Depending on the current structure of the hierarchy reduces the program's flexibility.
- Minimize the number of segments your program locks. You may need to take checkpoints to release the locks on updated segments and the lock on the current database record for each PCB your program uses. Each PCB used by your program has the current database record locked at share or update level. If this lock is no longer required, issuing the `GU` call, qualified at the root level with a greater-than operator for a key of X'FF' (high values), releases the current lock without acquiring a new lock.

  Do not use the minimization technique if you use a randomizer that puts high values at the end of the database and you use secondary indexes. If there is another root beyond the supposed high value key, IMS returns a GE to allow the application to determine the next step. A secondary index might not work because the hierarchical structure is inverted, and although the key is past the last root in the index, it might not be past the last root in the database.

  Using PCBs with a processing option of get (G) results in locks for the PCB at share level. This allows other programs that use the get processing option to concurrently access the same database record. Using a PCB with a processing option that allows updates (I, R, or D) results in locks for the PCB at update level. This does not allow any other program to concurrently access the same database record.

Related concepts:

"Reserving segments for the exclusive use of your program" on page 291

Segment search arguments (SSAs)

Segment search arguments (SSAs) specify information for IMS to use in processing a DL/I call. Regardless of the datatype for the field specified in a SSA, the SSA treats the field as a binary type and does a binary comparison.

A DL/I call with one or more SSAs is a *qualified call*, and a DL/I call without SSAs is an *unqualified call*.

Unqualified SSAs
: Contains only a segment name.

Qualified SSAs
: Includes one or more qualification statements that name a segment occurrence. The C command and a segment occurrence's concatenated key can be substituted for a qualification statement.

You can use SSA to select segments by name and to specify search criteria for specific segments. Specific segments are described by adding qualification statements to the DL/I call. You can further qualify your calls by using command codes.

Unqualified SSAs

An unqualified SSA gives the name of the segment type that you want to access. In an unqualified SSA, the segment name field is 8 bytes and must be followed by a 1-byte blank. If the actual segment name is fewer than 8 bytes long, it must be padded to the right with blanks. An example of an unqualified SSA follows:

```
PATIENT␣␣
```

Qualified SSAs

To qualify an SSA, you can use either a field or the sequence field of a virtual child. A qualified SSA describes the segment occurrence that you want to access. This description is called a qualification statement and has three parts. The following table shows the structure of a qualified SSA.

Table 31. Qualified SSA structure

SSA Component	Field Length
Segment ame	8
(	1
Field name	8
Relative operator	2
Field value	Variable
)	1

Using a qualification statement enables you to give IMS information about the particular segment occurrence that you are looking for. You do this by giving IMS the name of a field within the segment and the value of the field you are looking for. The field and the value are connected by a relational operator (`R.O.` in the previous table) which tells IMS how you want the two compared. For example, to access the PATIENT segment with the value 10460 in the PATNO field, you could use this SSA:

```
PATIENT␣(PATNO␣␣=␣10460)
```

The qualification statement is enclosed in parentheses. The first field contains the name of the field (`Fld Name` in the previous table) that you want IMS to use in searching for the segment. The second field contains a relational operator. The relational operator can be any one of the following:

- Equal, represented as

  =␣

  ␣=

  EQ
- Greater than, represented as

  >␣

  ␣>

  GT
- Less than, represented as

<△
△<
LT

- Greater than or equal to, represented as

  >=
  =>
  GE

- Less than or equal to, represented as

  <=
  =<
  LE

- Not equal to, represented as

  ¬=
  =¬
  NE

The third field (`Fld Value` in the previous table) contains the value that you want IMS to use as the comparative value. The length of `Fld Value` must be the same length as the field specified by `Fld Name`.

You can use more than one qualification statement in an SSA. Special cases exist, such as in a virtual logical child segment when the sequence field consists of multiple fields.

Sequence fields of a virtual logical child

As a general rule, a segment can have only one sequence field. However, in the case of the virtual logical-child segment type, multiple FIELD statements can be used to define a noncontiguous sequence field.

When specifying the sequence field for a virtual logical child segment, if the field is not contiguous, the length of the field named in the SSA is the concatenated length of the specified field plus all succeeding sequence fields. The following figure shows a segment with a noncontiguous sequence field.

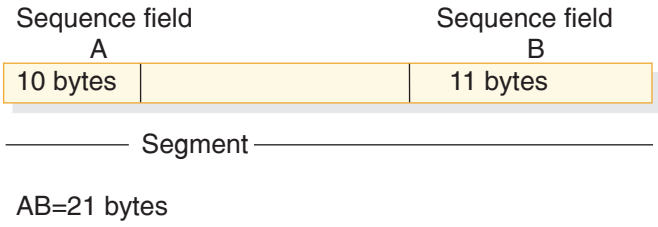


Figure 47. Segment with a noncontiguous sequence field

If the first sequence field is not included in a "scattered" sequence field in an SSA, IMS treats the argument as a data field specification, rather than as a sequence field.

Related reading: For more information on the virtual logical child segment, refer to *IMS Version 11 Database Administration*.

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

SSA guidelines

Using SSAs can simplify your programming, because the more information you can give IMS to do the searching for you, the less program logic you need to analyze and compare segments in your program.

Using SSAs does not necessarily reduce system overhead, such as internal logic and I/Os, required to obtain a specific segment. To locate a particular segment without using SSAs, you can issue DL/I calls and include program logic to examine key fields until you find the segment you want. By using SSAs in your DL/I calls, you can reduce the number of DL/I calls that are issued and the program logic needed to examine key fields. When you use SSAs, IMS does this work for you.

Recommendations:

- Use qualified calls with qualified SSAs whenever possible. SSAs act as filters, returning only the segments your program requires. This reduces the number of calls your program makes, which provides better performance. It also provides better documentation of your program. Qualified SSAs are particularly useful when adding segments with insert calls. They ensure that the segments are inserted where you want them to go.
- For the root segment, specify the key field and an equal relational operator, if possible. Using a key field with an equal-to, equal-to-or-greater-than, or greater-than operator lets IMS go directly to the root segment.
- For dependent segments, it is desirable to use the key field in the SSA, although it is not as important as at the root level. Using the key field and an equal-to operator lets IMS stop the search at that level when a higher key value is encountered. Otherwise IMS must search through all occurrences of the segment type under its established parent in order to determine whether a particular segment exists.
- If you often must search for a segment using a field other than the key field, consider putting a secondary index on the field.

For example, suppose you want to find the record for a patient by the name of "Ellen Carter". As a reminder, the patient segment in the examples contains three fields: the patient number, which is the key field; the patient name; and the patient address. The fact that patient number is the key field means that IMS stores the patient segments in order of their patient numbers. The best way to get the record for "Ellen Carter" is to supply her patient number in the SSA. If her number is 09000, your program uses this call and SSA:

```
GU&$tab;PATIENT⌂
(PATNO⌂⌂⌂
=⌂
09000)
```

If your program supplies an invalid number, or if someone has deleted Ellen Carter's record from the database, IMS does not need to search through all the PATIENT occurrences to determine that the segment does not exist.

However, if your program does not have the number and must give the name instead, IMS must search through all the patient segments and read each patient name field until it finds "Ellen Carter" or until it reaches the end of the patient segments.

Related concepts:

Chapter 17, “Secondary indexing and logical relationships,” on page 293

Multiple qualification statements

When you use a qualification statement, you can do more than give IMS a field value with which to compare the fields of segments in the database. You can give several field values to establish limits for the fields you want IMS to compare.

You can use a maximum of 1024 qualification statements on a call.

Connect the qualification statements with one of the Boolean operators. You can indicate to IMS that you are looking for a value that, for example, is greater than A **and** less than B, or you can indicate that you are looking for a value that is equal to A **or** greater than B. The Boolean operators are:

Logical AND
: For a segment to satisfy this request, the segment must satisfy both qualification statements that are connected with the logical AND (coded * or &).

Logical OR
: For a segment to satisfy this request, the segment can satisfy either of the qualification statements that are connected with the logical OR (coded + or |).

One more Boolean operator exists and is called the independent AND. Use it only with secondary indexes.

For a segment to satisfy multiple qualification statements, the segment must satisfy a set of qualification statements. A set is a number of qualification statements that are joined by an AND. To satisfy a set, a segment must satisfy each of the qualification statements within that set. Each OR starts a new set of qualification statements. When processing multiple qualification statements, IMS reads them left to right and processes them in that order.

When you include multiple qualification statements for a root segment, the fields you name in the qualification statements affect the range of roots that IMS examines to satisfy the call. DL/I examines the qualification statements to determine the minimum acceptable key value.

If one or more of the sets do not include at least one statement that is qualified on the key field with an operator of equal-to, greater-than, or equal-to-or-greater-than, IMS starts at the first root of the database and searches for a root that meets the qualification.

If each set contains at least one statement that is qualified on the key field with an equal-to, greater-than, or equal-to-or-greater-than operator, IMS uses the lowest of these keys as the starting place for its search. After establishing the starting position for the search, IMS processes the call by searching forward sequentially in the database, similar to the way it processes `GN` calls. IMS examines each root it encounters to determine whether the root satisfies a set of qualification statements. IMS also examines the qualification statements to determine the maximum acceptable key value.

If one or more of the sets do not include at least one statement that is qualified on the key field with an operator of equal-to, less-than-or-equal-to, or less-than, IMS determines that no maximum key value exists. If each set contains at least one

statement that is qualified on the key field with an equal-to, less-than, or equal-to-or-less-than operator, IMS uses the maximum of these keys to determine when the search stops.

IMS continues the search until it satisfies the call, encounters the end of the database, or finds a key value that exceeds the maximum. If no maximum key value is found, the search continues until IMS satisfies the call or encounters the end of the database.

Examples: Shown below are cases of SSAs used at the root level:

```
ROOTKEY␣
=␣10&FIELDB␣
␣=XYZ+ROOTKEY␣
␣=10&FIELDB␣
␣
=ABC
```

In this case, the minimum and maximum key is 10. This means that IMS starts searching with key 10 and stops when it encounters the first key greater than 10. To satisfy the SSA, the ROOTKEY field must be equal to 10, and FIELDB must be equal to either ABC or XYZ.

```
ROOTKEY␣
=>10&ROOTKEY␣
<20
```

In this case, the minimum key is 10 and the maximum key is 20. Keys in the range of 10 to 20 satisfy the SSA. IMS stops the search when it encounters the first key greater than 20.

```
ROOTKEY␣
10&ROOTKEY␣
=<20+ROOTKEY␣
=>110&ROOTKEY␣
=<120
```

In this case, the minimum key is 10 and the maximum key is 120. Keys in the range of 10 to 20 and 110 to 120 satisfy the call. IMS stops the search when it encounters the first key greater than 120. IMS does not scan from 20 to 110 but skips forward (using the index for HIDAM or PHIDAM) from 20 to 110. Because of this, you can use ranges for more efficient program operation.

When you use multiple qualification statement segments that are part of logical relationships, additional considerations exist.

Related concepts:

"Multiple qualification statements with secondary indexes" on page 294

Example of how to use multiple qualification statements

The following example shows how you can use multiple qualification statements.

Given the sample Medical database, we want to answer the following question:

> Did we see patient number 04120 during 1992?

To find the answer to this question, you need to give IMS more than the patient's name; you want IMS to search through the ILLNESS segments for that patient, read each one, and return any that have a date in 1992. The call you would issue to do this is:

```
GU   PATIENTb
(PATNObbb
EQ04120)
     ILLNESSb
(ILLDATEb
>=19920101&ILLDATEb
<=19921231)
```

In other words, you want IMS to return any ILLNESS segment occurrences under patient number 04120 that have a date on or after January 1, 1992, and on or before December 31, 1992, joined with an AND connector. Suppose you wanted to answer the following request:

> Did we see Judy Jennison during January of 1992 or during July of 1992? Her patient number is 05682.

You could issue a GU call with the following SSAs:

```
GU   PATIENTb
PATNOb
EQ05682)
     ILLNESSb
(ILLDATEb
>=19920101&ILLDATEb
<=19920131|
               ILLDATEb
>=19920701&ILLDATEb
<=19920731)
```

To satisfy this request, the value for ILLDATE must satisfy either of the two sets. IMS returns any ILLNESS segment occurrences for the month of January 1992, or for the month of July 1992.

Multiple qualification statements for HDAM, PHDAM, or DEDB

For HDAM (Hierarchical Direct Access Method), PHDAM (partitioned HDAM), or data entry database (DEDB) organizations, a randomizing exit routine usually does not store the root keys in ascending key sequence. For these organizations, IMS determines the minimum and maximum key values. The minimum key value is passed to the randomizing exit routine, which determines the starting anchor point.

The first root off this anchor is the starting point for the search. When IMS encounters a key that exceeds the maximum key value, IMS terminates the search with a GE status code. If the randomizing routine randomized so that the keys are stored in ascending key sequence, a call for a range of keys will return all of the keys in the range. However, if the randomizing routine did not randomize into key sequence, the call does not return all keys in the requested range. Therefore, use calls for a range of key values only when the keys are in ascending sequence (when the organization is HDAM, PHDAM, or DEDB).

Recommendations:

- When the organization is HDAM, PHDAM, or DEDB, use calls for a range of key values only when the keys are in ascending sequence.
- When the organization is HDAM, PHDAM or DEDB, do not use calls that allow a range of values at the root level.

While not recommended, a sequential search of the database can be accomplished with the use of command codes A and G when making GN/GHN database calls. Command code A will clear positioning and cause the call to start at the beginning

of the database. Command code G will prevent randomization and cause a sequential search of the database when used with SSAs that specify a range of values at the root level. The returned segments may not be in sequential order depending on how they were randomized.

To search the database sequentially, you can use the use the following segment search argument (SSA) together with SSAs that specify a range of values at the root level.

```
key field > hex zeros & key field < all f's key
```

The returned segments may not be in sequential order depending on how they were randomized.

For more details about HDAM or PHDAM databases, see *IMS Version 11 Database Administration*.

SSAs and command codes

SSAs can also include one or more command codes, which can change and extend the functions of DL/I calls.

For information on command codes, see the topic "General Command Codes for DL/I Calls" in *IMS Version 11 Application Programming APIs*.

IMS always returns the lowest segment in the path to your I/O area. If your program codes a D command code in an SSA, IMS also returns the segment described by that SSA. A call that uses the D command code is called a *path call*.

For example, suppose your program codes a D command code on a `GU` call that retrieves segment F and all segments in the path to F in the hierarchy shown in the following figure.

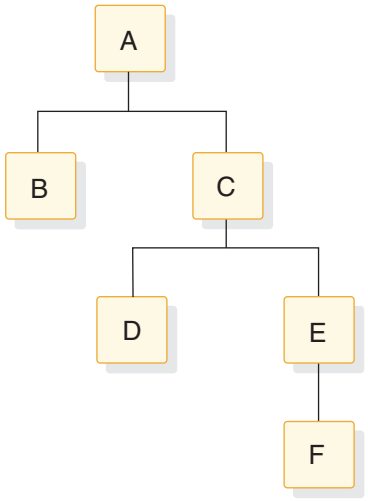


Figure 48. D command code example

The call function and the SSAs for the call look like this:

```
GU   A△△△△△△
*D
          C△△△△△△
```

```
*D
          E^^^^^^^
          F^^^^^^^
```

A command code consists of one letter. Code the command codes in the SSA after the segment name field. Separate the segment name field and the command code with an asterisk, as shown in the following table.

Table 32. Unqualified SSA with command code

SSA Component	Field Length
Seg Name	8
*	1
Cmd Code	Variable
b	1

Your program can use command codes in both qualified and unqualified SSAs. However, command codes cannot be used by MSDB calls. If the command codes are not followed by qualification statements, they must each be followed by a 1-byte blank. If the command codes are followed by qualification statements, do not use the blank. The left parenthesis of the qualification statement follows the command code instead, as indicated in the following table.

Table 33. Qualified SSA with command code

SSA Component	Field Length
Seg Name	8
*	1
Cmd Code	Variable
(	1
Fld Name	8
R.O.	2
Fld Value	Variable
)	1

If your program uses command codes to manage subset pointers in a DEDB, enter the number of the subset pointer immediately after the command code. Subset pointers are a means of dividing a chain of segment occurrences under the same parent into two or more groups or subsets. Your program can define as many as eight subset pointers for any segment type. Using an application program, your program can then manage these subset pointers.

Related concepts:

"Processing Fast Path DEDBs with subset pointer command codes" on page 328

Considerations for coding DL/I calls and data areas

If you have made all the design decisions about your program, coding the program is a matter of implementing the decisions that you have made. In addition to knowing the design and processing logic for your program, you need to know about the data that your program is processing, the PCBs it references, and the segment formats in the hierarchies your program processes.

You can use the following list as a checklist to make sure you are not missing any information. If you are missing information about data, IMS options being used in the application program, or segment layouts and the application program's data structures, obtain this information from the DBA or the equivalent specialist at your installation. Be aware of the programming standards and conventions that have been established at your installation.

Program design considerations:

- The sequence of calls for your program.
- The format of each call:
  - Does the call include any SSAs?
  - If so, are they qualified or unqualified?
  - Does the call contain any command codes?
- The processing logic for the program.
- The routine the program uses to check the status code after each call.
- The error routine the program uses.

Checkpoint considerations:

- The type of checkpoint call to use (basic or symbolic).
- The identification to assign to each checkpoint call, regardless of whether the Checkpoint call is basic or symbolic.
- If you are going to use the symbolic checkpoint call, which areas of your program to checkpoint.

Segment considerations:

- Whether the segment is fixed length or variable length.
- The length of the segment (the maximum length, if the segment is variable length).
- The names of the fields that each segment contains.
- Whether the segment has a key field. If it does, is the key field unique or non-unique? If it does not, what sequencing rule has been defined for it? (A segment's key field is defined in the SEQ keyword of the FIELD statement in the DBD. The sequencing rule is defined in the RULES keyword of the SEGM statement in the DBD.)
- The segment's field layouts:
  - The byte location of each field.
  - The length of each field.
  - The format of each field.

Data structure considerations:

- Each data structure your program processes has been defined in a DB PCB. All of the PCBs your program references are part of a PSB for your application program. You need to know the order in which the PCBs are defined in the PSB.
- The layout of each of the data structures your program processes.
- Whether multiple or single positioning has been specified for each data structure. This is specified in the `POS` keyword of the PCB statement during PSB generation.
- Whether any data structures use multiple DB PCBs.

Preparing to run your CICS DL/I call program

You must perform several steps before you run your CICS DL/I call program.

Refer to the appropriate CICS reference information:

- For information on translating, compiling, and binding your CICS online program, see the description of installing application programs in *CICS Transaction Server for z/OS CICS System Definition Guide*.
- For information on which compiler options should be used for a CICS online program, as well as for CICS considerations when converting a CICS online COBOL program with DL/I calls to Enterprise COBOL, see *CICS Transaction Server for z/OS CICS Application Programming Guide*.

Examples of how to code DL/I calls and data areas

You can code DL/I calls and data areas in assembler language, C, COBOL, Pascal, Java, and PL/I.

Coding a batch program in assembler language

The following code example shows how to write an IMS program to access the IMS database in assembler language.

The numbers to the right of the program refer to the notes that follow the program. This kind of program can run as a batch program or as a batch-oriented BMP.

Sample assembler language program

```
PGMSTART CSECT                                                          NOTES
*               EQUATE REGISTERS                                         1
*   USEAGE OF REGISTERS
R1       EQU   1              ORIGINAL PCBLIST ADDRESS
R2       EQU   2              PCBLIST ADDRESS1
R5       EQU   5              PCB ADDRESSS
R12      EQU   12             BASE ADDRESS
R13      EQU   13             SAVE AREA ADDRESS
R14      EQU   14
R15      EQU   15
*
         USING PGMSTART,R12   BASE REGISTER ESTABLISHED                  2
         SAVE  (14,12)        SAVE REGISTERS
         LR    12,15          LOAD REGISTERS
         ST    R13,SAVEAREA+4 SAVE AREA CHAINING
         LA    R13,SAVEAREA   NEW SAVE AREA
         USING PCBLIST,R2     MAP INPUT PARAMETER LIST
         USING PCBNAME,R5     MAP DB PCB
         LR    R2,R1          SAVE INPUT PCB LIST IN REG 2
         L     R5,PCBDETA     LOAD DETAIL PCB ADDRESS
         LA    R5,0(R5)       REMOVE HIGH ORDER END OF LIST FLAG         3
         CALL  ASMTDLI,(GU,(R5),DETSEGIO,SSANAME),VL                     4
*
*
         L     R5,PCBMSTA     LOAD MASTER PCB ADDRESS
         CALL  ASMTDLI,(GHU,(R5),MSTSEGIO,SSAU),VL                       5
*
*
         CALL  ASMTDLI,(GHN,(R5),MSTSEGIO),VL                            6
*
*
         CALL  ASMTDLI,(REPL,(R5),MSTSEGIO),VL
*
```

```
*
         L     R13,4(R13)     RESTORE SAVE AREA
         RETURN (14,12)     RETURN BACK                                    7
*
*     FUNCTION CODES USED
*
GU       DC    CL4'GU'
GHU      DC    CL4'GHU'
GHN      DC    CL4'GHN'
REPL     DC    CL4'REPL'                                                   8
*
*       SSAS
*
SSANAME  DS    0C
         DC    CL8'ROOTDET'
         DC    CL1'('
         DC    CL8'KEYDET'                                                 9
         DC    CL2' ='
NAME     DC    CL5'  '
         DC    C')'
*
SSAU     DC    CL9'ROOTMST'*
MSTSEGIO DC    CL100' '
DETSEGIO DC    CL100' '
SAVEAREA DC    18F'0'
*                                                                         10
PCBLIST  DSECT
PCBIO    DS    A              ADDRESS OF I/O PCB
PCBMSTA  DS    A              ADDRESS OF MASTER PCB
PCBDETA  DS    A              ADDRESS OF DETAIL PCB                       11
*
PCBNAME  DSECT
DBPCBDBD DS    CL8            DBD NAME
DBPCBLEV DS    CL2            LEVEL FEEDBACK
DBPCBSTC DS    CL2            STATUS CODES
DBPCBPRO DS    CL4            PROC OPTIONS
DBPCBRSV DS    F              RESERVED
DBPCBSFD DS    CL8            SEGMENT NAME FEEDBACK
DBPCBMKL DS    F              LENGTH OF KEY FEEDBACK
DBPCBNSS DS    F              NUMBER OF SENSITIVE SEGMENTS IN PCB
DBPCBKFD DS    C              KEY FEEDBACK AREA
         END   PGMSTART
```

Note:

1. The entry point to an assembler language program can have any name. Also, you can substitute CBLTDLI for ASMTDLI in any of the calls.
2. When IMS passes control to the application program, register 1 contains the address of a variable-length fullword parameter list. Each word in this list contains the address of a PCB that the application program must save. The high-order byte of the last word in the parameter list has the 0 bit set to a value of 1 which indicates the end of the list. The application program subsequently uses these addresses when it executes DL/I calls.
3. The program loads the address of the DETAIL DB PCB.
4. The program issues a GU call to the DETAIL database using a qualified SSA (SSANAME).
5. The program loads the address of the HALDB master PCB.
6. The next three calls that the program issues are to the HALDB master. The first is a GHU call that uses an unqualified SSA. The second is an unqualified GHN call. The REPL call replaces the segment retrieved using the GHN call with the segment in the MSTSEGIO area.

You can use the *parmcount* parameter in DL/I calls in assembler language instead of the VL parameter, except for in the call to the sample status-code error routine.

7. The RETURN statement loads IMS registers and returns control to IMS.
8. The call functions are defined as four-character constants.
9. The program defines each part of the SSA separately so that it can modify the SSA's fields.
10. The program must define an I/O area that is large enough to contain the largest segment it is to retrieve or insert (or the largest path of segments if the program uses the D command code). This program's I/O areas are 100 bytes each.
11. A fullword must be defined for each PCB. The assembler language program can access status codes after a DL/I call by using the DB PCB base addresses.

    This example assumes that an I/O PCB was passed to the application program. If the program is a batch program, CMPAT=YES must be specified on the PSBGEN statement of PSBGEN so that the I/O PCB is included. Because the I/O PCB is required for a batch program to make system service calls, CMPAT=YES should always be specified.

Restriction: The IMS language interface module (DFSLI000) must be bound to the compiled assembler language program.

Coding a CICS online program in assembler language

The following code example in assembler language shows how you define and establish addressability to the UIB.

The numbers to the right of the program refer to the notes that follow the program. This program can run in a CICS environment using DBCTL.

Sample call-level assembler language program (CICS online)

```
PGMSTART DSECT                                          NOTES
UIBPTR   DS    F
IOAREA   DS    0CL40                                      1
AREA1    DS    CL3
AREA2    DS    CL37
         DLIUIB
         USING UIB,8                                      2
PCBPTRS  DSECT
*        PSB ADDRESS LIST
PCB1PTR  DS    F
PCB1     DSECT
         USING PCB1,6                                   3
DBPC1DBD DS    CL8
DBPC1LEV DS    CL2
DBPC1STC DS    CL2
DBPC1PRO DS    CL4
DBPC1RSV DS    F
DBPC1SFD DS    CL8
DBPC1MKL DS    F
DBPC1NSS DS    F
DBPC1KFD DS    0CL256
DBPC1NM  DS    0CL12
DBPC1NMA DS    0CL14
DBPC1NMP DS    CL17
ASMUIB   CSECT
         B     SKIP
PSBNAME  DC    CL8'ASMPSB'
PCBFUN   DC    CL4'PCB'
```

```
REPLFUN  DC    CL4'REPL'
TERMFUN  DC    CL4'TERM'
GHUFUN   DC    CL4'GHU'
SSA1     DC    CL9'AAAA4444'
GOODRC   DC    XL1'00'
GOODSC   DC    CL2'  '
SKIP     DS    0H                                                    4
*              SCHEDULE PSB AND OBTAIN PCB ADDRESSES
         CALLDLI ASMTDLI,(PCBFUN,PSBNAME,UIBPTR)
         L     8,UIBPTR                                              5
         CLC   UIBFCTR,X'00'
         BNE   ERROR1
*              GET PSB ADDRESS LIST
         L     4,UIBPCBAL
         USING PCBPTRS,4
*              GET ADDRESS OF FIRST PCB IN LIST
         L     6,PCB1PTR
*              ISSUE DL/I CALL: GET A UNIQUE SEGMENT
         CALLDLI ASMTDLI,(GHUFUN,PCB1,IOAREA,SSA1)                   6
         CLC   UIBFCTR,GOODRC
         BNE   ERROR2
         CLC   DBPC1STC,GOODSC
         BNE   ERROR3                                                7
*              PERFORM SEGMENT UPDATE ACTIVITY
         MVC   AREA1,.......
         MVC   AREA2,.......
*              ISSUE DL/I CALL: REPLACE SEGMENT AT CURRENT POSITION
         CALLDLI ASMTDLI,(REPLFUN,PCB1,IOAREA,SSA1)                  8
         CLC   UIBFCTR,GOODRC
         BNE   ERROR4
         CLC   DBPC1STC,GOODSC
         B     TERM
ERROR1   DS    0H
*              INSERT ERROR DIAGNOSTIC CODE
         B     TERM
ERROR2   DS    0H
*              INSERT ERROR DIAGNOSTIC CODE
         B     TERM
ERROR3   DS    0H
*              INSERT ERROR DIAGNOSTIC CODE
         B     TERM
ERROR4   DS    0H
*              INSERT ERROR DIAGNOSTIC CODE
ERROR5   DS    0H
*              INSERT ERROR DIAGNOSTIC CODE
         B     TERM
TERM     DS    0H
*              RELEASE THE PSB
         CALLDLI ASMDLI, (TERMFUN)
         EXEC CICS RETURN
         END   ASMUIB                                              9,10
```

Note:

1. The program must define an I/O area that is large enough to contain the largest segment it is to retrieve or insert (or the largest path of segments if the program uses the D command code).
2. The DLIUIB statement copies the UIB DSECT.
3. A fullword must be defined for each DB PCB. The assembler language program can access status codes after a DL/I call by using the DB PCB base addresses.
4. This is an unqualified SSA. For qualified SSA, define each part of the SSA separately so that the program can modify the fields of the SSA.
5. This call schedules the PSB and obtains the PSB address.

6. This call retrieves a segment from the database.

   CICS online assembler language programs use the CALLDLI macro, instead of the call statement, to access DL/I databases. This macro is similar to the call statement. It looks like this:

   ```
   CALLDLI ASMTDLI,(function,PCB-name,ioarea, SSA1,...SSAn),VL
   ```

7. CICS online programs must check the return code in the UIB before checking the status code in the DB PCB.
8. The REPL call replaces the data in the segment that was retrieved by the most recent Get Hold call. The data is replaced by the contents of the I/O area referenced in the call.
9. This call releases the PSB.
10. The RETURN statement loads IMS registers and returns control to IMS.

Related reading: For more information on installing CICS application programs, see *CICS Transaction Server for z/OS CICS Application Programming Reference*.

Related reference:

"Specifying the UIB (CICS online programs only)" on page 237

Coding a batch program in C language

The following code example shows how to write an IMS program to access the IMS database in C language.

The numbers to the right of the program refer to the notes that follow the program.

Sample C language program

```
#pragma runopts(env(IMS),plist(IMS))                                    NOTES
#include <ims.h>
#include <stdio.h>                                                        1
main() {                                                                  2
/*                                                          */
/*             descriptive statements                       */
/*                                                          */
 IO_PCB_TYPE *IO_PCB = (IO_PCB_TYPE*)PCBLIST[0];
 struct {PCB_STRUCT(10)} *mast_PCB   = __pcblist[1];
struct {PCB_STRUCT(20)} *detail_PCB = __pcblist[2];                       3
const static char func_GU[4]   = "GU  ";
const static char func_GN[4]   = "GN  ";
const static char func_GHU[4]  = "GHU ";
const static char func_GHN[4]    = "GHN ";
const static char func_GNP[4]    = "GNP ";                              4
const static char func_GHNP[4] = "GHNP";
const static char func_ISRT[4] = "ISRT";
const static char func_REPL[4] = "REPL";
const static char func_DLET[4] = "DLET";
char qual_ssa[8+1+8+2+6+1+1]; /* initialized by sprintf                  5
                              /*below.  See the  */
                              /*explanation for  */
                              /*sprintf in note 7 for the */
                              /*meanings of 8,1,8,2,6,1 --*/
                              /*the final 1 is for the    */
                              /*trailing '\0' of string */
static const char unqual_ssa[]= "NAME     ");
                              /* 12345678_ */
struct {
          ---
          ---
          ---
       } mast_seg_io_area;
```

```
struct {
          ___
          ___                                                    6
          ___
       } det_seg_io_area;
/*                                                  */
/*        Initialize the qualifier                  */
/*                                                  */

    sprintf(qual_ssa,
           "8.8s(8.8s6.6s)",
           "ROOT", "KEY", "=", "vvvvv");                         7
/*                                                  */
/*            Main part of C batch program          */
/*                                                  */
     ctdli(func_GU, detail_PCB,
          &det_seg_io_area,qual_ssa);                           8

     ctdli(func_GHU, mast_PCB,
          &mast_seg_io_area,qual_ssa);                          9

     ctdli(func_GHN, mast_PCB,
          &mast_seg_io_area);                                   10

     ctdli(func_REPL, mast_PCB,
          &mast_seg_io_area;                                    11
}                                                               12
```

Note:

1. The **env(IMS)** establishes the correct operating environment and the **plist(IMS)** establishes the correct parameter list when invoked under IMS. The **ims.h** header file contains declarations for PCB layouts, **__pcblist**, and the **ctdli** routine. The PCB layouts define masks for the PCBs that the program uses as structures. These definitions make it possible for the program to check fields in the PCBs.

   The **stdio.h** header file contains declarations for **sprintf** (used to build up the SSA).
2. After IMS has loaded the application program's PSB, IMS gives control to the application program through this entry point.
3. The C run-time sets up the **__pcblist** values. The order in which you refer to the PCBs must be the same order in which they have been defined in the PSB. (Values other than “10” and “20” can be used, according to the actual key lengths needed.) These declarations can be done using macros, such as:

   ```
   #define IO_PCB (IO_PCB_TYPE *) (__pcblist[0])
   #define mast_PCB   (__pcblist[1])
   #define detail_PCB (__pcblist[2])
   ```

   This example assumes that an I/O PCB was passed to the application program. When the program is a batch program, CMPAT=YES must be specified on the PSBGEN statement of PSBGEN so that the I/O PCB is included. Because the I/O PCB is required for a batch program to make system service calls, CMPAT=YES should always be specified for batch programs.
4. Each of these areas defines one of the call functions used by the batch program. Each character string is defined as four alphanumeric characters, with a value assigned for each function. (If the **[4]**s had been left out, 5 bytes would have been reserved for each constant.) You can define other constants in the same way. Also, you can store standard definitions in a source library and include them by using a **#include** directive.

Instead, you can define these by macros, although each string would have a trailing null (**'\0'**).

5. The SSA is put into a string (see note 7). You can define a structure, as in COBOL, PL/I, or Pascal, but using **sprintf** is more convenient. (Remember that C strings have trailing nulls that cannot be passed to IMS.) Note that the string is 1 byte longer than required by IMS to contain the trailing null, which is ignored by IMS. Note also that the numbers in brackets assume that six fields in the SSA are equal to these lengths.
6. The I/O areas that will be used to pass segments to and from the database are defined as structures.
7. The **sprintf** function is used to fill in the SSA. The "**%-8.8s**" format means "a left-justified string of exactly eight positions". The "**%2.2s**" format means "a right-justified string of exactly two positions".

   Because the **ROOT** and **KEY** parts do not change, this can also be coded:

   ```
   sprintf(qual_ssa,
           "ROOT    (KEY      =%-6.6s)", "vvvvv");
         /* 12345678 12345678 */
   ```

8. This call retrieves data from the database. It contains a qualified SSA. Before you can issue a call that uses a qualified SSA, initialize the data field of the SSA. Before you can issue a call that uses an unqualified SSA, initialize the segment name field. Unlike the COBOL, PL/I, and Pascal interface routines, **ctdli** also returns the status code as its result. (Blank is translated to 0.) So, you can code:

   ```
   switch (ctdli(....)) {
       case 0: ... /* everything ok */

               break;
       case 'AB': ....

               break;
       case 'IX': ...

               break;
       default:
   }
   ```

   You can pass only the PCB pointer for DL/I calls in a C program.
9. This is another call with a qualified SSA.
10. This call is an unqualified call that retrieves data from the database. Because it is a Get Hold call, it can be followed by `REPL` or `DLET`.
11. The `REPL` call replaces the data in the segment that was retrieved by the most recent Get Hold call. The data is replaced by the contents of the I/O area that is referenced in the call.
12. The end of the **main** routine (which can be done by a **return** statement or **exit** call) returns control to IMS.

Restriction: IMS provides a language interface module (DFSLI000) that is an interface between IMS and the C language. This module must be made available to the application program at bind time.

Coding a batch program in COBOL

The following code example shows how to write an IMS program to access the IMS database in COBOL.

The numbers to the right of the program refer to the notes that follow the program. This kind of program can run as a batch program or as a batch-oriented BMP.

Sample COBOL program

```
       Identification Division.
       Program-ID. BATCOBOL.
       Environment Division.
       Data Division.
       Working-Storage Section.
         01 Func-Codes.
            05 Func-GU          Picture XXXX Value 'GU  '.
            05 Func-GHU         Picture XXXX Value 'GHU '.
            05 Func-GN          Picture XXXX Value 'GHN '.
            05 Func-GHN         Picture XXXX Value 'GHN '.
            05 Func-GNP         Picture XXXX Value 'GNP '.
            05 Func-GHNP        Picture XXXX Value 'GHNP'.
            05 Func-REPL        Picture XXXX Value 'REPL'.
            05 Func-ISRT        Picture XXXX Value 'ISRT'.
            05 Func-DLET        Picture XXXX Value 'DLET'.
            05 Parmcount        Picture S9(5) Value +4 Comp-5.
         01 Unqual-SSA.
            05 Seg-Name         Picture X(08) Value  '        '.
            05 Filler           Picture X     Value  ' '.
         01 Qual-SSA-Mast.
            05 Seg-Name-M       Picture X(08) Value 'ROOTMast'.
            05 Begin-Paren-M    Picture X     Value '('.
            05 Key-Name-M       Picture X(08) Value 'KeyMast '.
            05 Kel-Oper-M       Picture X(05) Value ' ='.
            05 Key-Value-M      Picture X(06) Value 'VVVVVV'.
            05 End-Paren-M      Picture X     Value ')'.
         01 Qual-SSA-Det.
            05 Seg-Name-D       Picture X(08) Value 'ROOTDET '.
            05 Begin-Paren-D    Picture X     Value '('.
            05 Key-Name-D       Picture X(08) Value 'KEYDET  '.
            05 Rel-Oper-D       Picture X(05) Value ' ='.
            05 Key-Value-D      Picture X(06) Value 'VVVVVV'.
            05 End-Paren-D      Picture X     Value ')'.
         01 Det-Seg-In.
            05 Data1            Picture X.
            05 Data2            Picture X.
        01 Mast-Seg-In.
            05 Data1            Picture X.
            05 Data2            Picture X.
       linkage section.
         01 IO-PCB.
            05 Filler           Picture X(10).
            05 IO-Status-Code   Picture XX.
            05 Filler           Picture X(20).
         01 DB-PCB-Mast.
            05 Mast-Dbd-Name    Picture X(8).
            05 Mast-Seg-Level   Picture XX.
            05 Mast-Status-Code Picture XX.
            05 Mast-Proc-Opt    Picture XXXX.
            05 Filler           Picture S9(5) Comp-5.
            05 Mast-Seg-Name    Picture X(8).
            05 Mast-Len-KFB     Picture S9(5) Comp-5.
            05 Mast-Nu-Senseg   Picture S9(5) Comp-5.
            05 Mast-Key-FB      Picture X(256).
         01 DB-PCB-Detail.
            05 Det-Dbd-Name     Picture X(8).
            05 Det-Seg-Level    Picture XX.
            05 Det-Status-Code  Picture XX.
            05 Det-Proc-Opt     Picture XXXX.
            05 Filler           Picture S9(5) Comp-5.
            05 Det-Seg-Name     Picture X(8).
```

```
           05 Det-Len-KFB      Picture S9(5) Comp-5.
           05 Det-Nu-Senseg    Picture S9(5) Comp-5.
           05 Det-Key-FB       Picture X(256).


       Procedure Division using IO-PCB DB-PCB-Mast DB-PCB-Detail.
           Call 'CBLTDLI' using Func-GU DB-PCB-Detail
                Det-seg-in Qual-SSA-Det.
            .
            .
           Call 'CBLTDLI' using Parmcount Func-ghu DB-PCB-Mast
                Mast-seg-in Qual-SSA-Mast.
            .
            .
           Call 'CBLTDLI' using Func-GHN DB-PCB-Mast
                Mast-seg-in.
            .
            .
           Call 'CBLTDLI' using Func-REPL DB-PCB-Mast
                Mast-seg-in.
            .
            .
           Goback.
```

Note:

1. You define each of the DL/I call functions the program uses with a 77-level or 01-level working storage entry. Each picture clause is defined as four alphanumeric characters and has a value assigned for each function. If you want to include the optional *parmcount* field, you can initialize count values for each type of call. You can also use a COBOL COPY statement to include these standard descriptions in the program.
2. A 9-byte area is set up for an unqualified SSA. Before the program issues a call that requires an unqualified SSA, it moves the segment name to this area. If a call requires two or more SSAs, you may need to define additional areas.
3. A 01-level working storage entry defines each qualified SSA that the application program uses. Qualified SSAs must be defined separately, because the values of the fields are different.
4. A 01-level working storage entry defines I/O areas that are used for passing segments to and from the database. You can further define I/O areas with sub-entries under the 01-level. You can use separate I/O areas for each segment type, or you can define one I/O area that you use for all segments.
5. A 01-level linkage section entry defines a mask for each of the PCBs that the program requires. The DB PCBs represent both input and output databases. After issuing each DL/I call, the program checks the status code through this linkage. You define each field in the DB PCB so that you can reference it in the program.
6. This is the standard procedure division statement of a batch program. After IMS has loaded the PSB for the program, IMS passes control to the application program. The PSB contains all the PCBs that are defined in the PSB. The coding of `USING` on the procedure division statement references each of the PCBs by the names that the program has used to define the PCB masks in the linkage section. The PCBs must be listed in the order in which they are defined in the PSB.

   The previous code example assumes that an I/O PCB was passed to the application program. When the program is a batch program, CMPAT=YES must be specified on the PSBGEN statement of PSBGEN so that the I/O PCB

is included. Because the I/O PCB is required for a batch program to make system service calls, CMPAT=YES should always be specified for batch programs.

The entry DLITCBL statement is only used in the main program. Do not use it in called programs.

7. This call retrieves data from the database by using a qualified SSA. Before issuing the call, the program must initialize the key or data value of the SSA so that it specifies the particular segment to be retrieved. The program should test the status code in the DB PCB that was referenced in the call immediately after issuing the call. You can include the *parmcount* parameter in DL/I calls in COBOL programs, except in the call to the sample status-code error routine. It is never required in COBOL.
8. This is another retrieval call that contains a qualified SSA.
9. This is an unqualified retrieval call.
10. The `REPL` call replaces the segment that was retrieved in the most recent Get Hold call. The segment is replaced with the contents of the I/O area that is referenced in the call (MAST-SEG-IN).
11. The program issues the GOBACK statement when it has finished processing.

Related reading: For information on how to use these procedures, see *IMS Version 11 System Definition*.

Binding COBOL code to the IMS language interface module

IMS supplies a language interface module (DFSLI000). This module must be bound to the batch program after the program has been compiled. It gives a common interface to IMS.

If you use the IMS-supplied procedures (IMSCOBOL or IMSCOBGO), IMS binds the language interface with the application program. IMSCOBOL is a two-step procedure that compiles and binds your program. IMSCOBGO is a three-step procedure that compiles, binds, and executes your program in an IMS batch region.

Coding a CICS online program in COBOL

The following code examples are skeleton online programs in Enterprise COBOL. They show examples of how to define and set up addressability to the UIB.

The numbers to the right of the programs refer to the notes that follow them. This kind of program can run in a CICS environment using DBCTL.

Sample COBOL program that can run in CICS

```
Identification Division.
Program-ID. CBLUIB.
Environment Division.
Data Division.
Working-Storage Section.
  01 Func-Codes.
     05  Psb-Name          Picture X(8) Value 'CBLPSB  '.
     05  Func-PCB          Picture X(4) Value 'PCB '.
     05  Func-TERM         Picture X(4) Value 'TERM'.                 1
     05  Func-GHU          Picture X(4) Value 'GHU '.
     05  Func-REPL         Picture X(4) Value 'REPL'.
     05  SSA1              Picture X(9) Value 'AAAA4444 '.
     05  Success-Message   Picture X(40).
     05  Good-Status-Code  Picture XX Value '  '.                      2
     05  good-return-code  Picture X Value low-Value.
```

```
 01  Message0.
     05  Message1           Picture X(38).                          3
     05  Message2           Picture XX.
 01  Dli-IO-Area.
     05  Area1              Picture X(3).
     05  Area2              Picture X(37).
       Procedure Division.
* Schedule the psb and address the uib
     Call 'CBLTDli' using Func-PCB Psb-Name                         4
          address of Dliuib.
     If Uibfctr is not equal low-Values then
* Insert error diagnostic code
        Exec CICS return end-exec
     End-if.
     Set address of pcb-addresses to pcbaddr.
* Issue DL/I Call: get a unique segment
     Set address of pcb1 to pcb-address-list(1).
     Call 'CBLTDli' using Func-GHU Pcb1                             5
          Dli-io-area ssa1.
     If uibfctr is not equal good-return-code then
* Insert error diagnostic code                                     6
        Exec CICS return end-Exec
     End-if.
     If pcb1-status-code is not equal good-status-code then
* Insert error diagnostic code
        Exec CICS return end-Exec
     End-if.
*    Perform segment update activity
     Move 'aaa' to area1.
     Move 'bbb' to area2.
*    Issue DL/I Call: replace segment at current position          7
     Call 'CBLTDli' using Func-REPL Pcb1
          Dli-io-area ssa1
     If uibfctr is not equal good-return-code then
* Insert error diagnostic code
        Exec CICS return end-Exec
     End-if.
     If pcb1-status-code is not equal good-status-code then
* Insert error diagnostic code
        Exec CICS return end-Exec
     End-if.
* Release the psb
     Call 'CBLTDli' using Func-TERM.
* Other application Function                                       8,9
     Exec CICS return end-Exec.
     Goback.
```

Note:

1. You define each of the DL/I call functions the program uses with a 77-level or 01-level working storage entry. Each picture clause is defined as four alphanumeric characters and has a value assigned for each function. If you want to include the optional *parmcount* field, initialize count values for each type of call. You can also use the COBOL COPY statement to include these standard descriptions in the program.
2. A 9-byte area is set up for an unqualified SSA. Before the program issues a call that requires an unqualified SSA, it can either initialize this area with the segment name or move the segment name to this area. If a call requires two or more SSAs, you may need to define additional areas.
3. An 01-level working storage entry defines I/O areas that are used for passing segments to and from the database. You can further define I/O areas with sub-entries under the 01-level. You can use separate I/O areas for each segment type, or you can define one I/O area that you use for all segments.

4. One PCB layout is defined in the linkage section. The PCB-ADDRESS-LIST occurs *n* times, where *n* is greater than or equal to the number of PCBs in the PSB.
5. The PCB call schedules a PSB for your program to use. The address of the DLIUIB parameter returns the address of DLIUIB.
6. This unqualified GHU call retrieves a segment from the database and places it in the I/O area that is referenced by the call. Before issuing the call, the program must initialize the key or data value of the SSA so that it specifies the particular segment to be retrieved.
7. CICS online programs should test the return code in the UIB before testing the status code in the DB PCB.
8. The REPL call replaces the segment that was retrieved in the most recent Get Hold call with the data that the program has placed in the I/O area.
9. The TERM call terminates the PSB the program scheduled earlier. This call is optional and is only issued if a sync point is desired prior to continued processing. The program issues the EXEC CICS RETURN statement when it has finished its processing. If this is a RETURN from the highest-level CICS program, a TERM call and sync point are internally generated by CICS.

Sample call-level OS/VS COBOL program for CICS online (obsolete with Enterprise COBOL)

```
      Identification Division.                                          NOTES
   Program-ID. CBLUIB.
       Environment Division.
       Data Division.
       Working-Storage Section.
         01 Func-Codes.
            05  Psb-Name          Picture X(8) Value 'CBLPSB  '.          1
            05  Func-PCB          Picture X(4) Value 'PCB '.
            05  Func-TERM         Picture X(4) Value 'TERM'.
            05  Func-GHU          Picture X(4) Value 'GHU '.
            05  Func-REPL         Picture X(4) Value 'REPL'.
            05  SSA1              Picture X(9) Value 'AAAA4444 '.         2
            05  Success-Message   Picture X(40).
            05  Good-Status-Code  Picture XX   Value '  '.
            05  Good-Return-Code  Picture X    Value low-Value.
       01  Message0.
            05  Message1          Picture X(38).
            05  Message2          Picture XX.
       01  Dli-IO-Area.                                                  3
            05  Area1             Picture X(3).
            05  Area2             Picture X(37).
       Linkage Section.                                                   4
       01  BllCells.
            05 FIller             Picture S9(8) Comp-5.
            05 Uib-Ptr            Picture S9(8) Comp-5.
            05 B-Pcb-Ptrs         Picture S9(8) Comp-5.
            05 Pcb1-Ptr           Picture S9(8) Comp-5.
            Copy DliUib.                                                 5,6
       01  Overlay-Dliuib Redefines Dliuib.
            05  Pcbaddr usage is pointer.
            05  Filler            Picture XX.
       01  Pcb-Ptrs.
            05 B-Pcb1-Ptr         Picture 9(8)  Comp-5.
       01  Pcb1.                                                         7
            05 Pcb1-Dbd-Name      Picture X(8).
            05 Pcb1-Seg-Level     Picture XX.
            05 Pcb1-Status-Code   Picture XX.
            05 Pcb1-PROC-OPT      Picture XXXX.
            05 FIller             Picture S9(5) Comp-5.
            05 Pcb1-Seg-Name      Picture X(8).
```

```
     05 Pcb1-Len-KFB        Picture S9(5) Comp-5.
     05 Pcb1-NU-ENSeg       Picture S9(5) Comp-5.
     05 Pcb1-KEY-FB         Picture X(256).
 Procedure Division.                                                    8
     Call 'CBLTDLI' using Func-PCB Psb-Name Uib-ptr.
     If Uibfctr is not equal low-values then
*       Insert error diagnostic Code
        Exec CICS Return end-Exec
     End-if.
     Move Uibpcbal to B-Pcb-Ptrs.
     Move B-Pcb1-Ptr to Pcb1-Ptr.

*       Issue DL/I Call: get a unique segment                          9
     Call 'CBLTDLI' using Func-GHU Pcb1
         Dli-io-area ssa1.
     Service reload Uib-ptr
     If Uibfctr is not equal Good-Return-Code then                     10
*       Insert error diagnostic Code
        Exec CICS Return end-Exec
     End-if.

     If Pcb1-Status-Code is not equal Good-Status-Code then
*       Insert error diagnostic Code
        Exec CICS Return end-Exec
     End-if.


*       Perform segment update activity
     Move 'aaa' to area1.
     Move 'bbb' to area2.
*       Issue DL/I Call: replace segment at current position          11
     Call 'CBLTDLI' using Func-REPL Pcb1
          Dli-io-area ssa1.
     If Uibfctr is not equal Good-Return-Code then
*       Insert error diagnostic Code
        Exec CICS Return end-Exec
     End-if.

     If Pcb1-Status-Code is not equal Good-Status-Code then
*       Insert error diagnostic Code
        Exec CICS Return end-Exec
     End-if.

*    Release the PSB
     Call 'CBLTDLI' using Func-TERM.                                12,13
       Exec CICS Return end-Exec.
```

Note:

1. You define each of the DL/I call functions the program uses with a 77-level or 01-level working storage entry. Each picture clause is defined as four alphanumeric characters and has a value assigned for each function. If you want to include the optional *parmcount* field, you can initialize count values for each type of call. You can also use the COBOL COPY statement to include these standard descriptions in the program.
2. A 9-byte area is set up for an unqualified SSA. Before the program issues a call that requires an unqualified SSA, it can either initialize this area with the segment name or move the segment name to this area. If a call requires two or more SSAs, you may need to define additional areas.
3. An 01-level working storage entry defines I/O areas that are used for passing segments to and from the database. You can further define I/O areas with 02-level entries. You can use separate I/O areas for each segment type, or you can define one I/O area to use for all segments.

4. The linkage section must start with a definition of this type to provide addressability to a parameter list that will contain the addresses of storage that is outside the working storage of the application program. The first 02-level definition is used by CICS to provide addressability to the other fields in the list. A one-to-one correspondence exists between the other 02-level names and the 01-level data definitions in the linkage section.
5. The COPY DLIUIB statement will be expanded.
6. The UIB returns the address of an area that contains the PCB addresses. The definition of PCB pointers is necessary to obtain the actual PCB addresses. Do not alter the addresses in the area.
7. The PCBs are defined in the linkage section.
8. The `PCB` call schedules a PSB for your program to use.
9. This unqualified `GHU` call retrieves a segment from the database and places it in the I/O area that is referenced by the call. Before issuing the call, the program must initialize the key or data value of the SSA so that it specifies the particular segment to be retrieved.
10. CICS online programs should test the return code in the UIB before testing the status code in the DB PCB.
11. The `REPL` call replaces the segment that was retrieved in the most recent Get Hold call with the data that the program has placed in the I/O area.
12. The `TERM` call terminates the PSB that the program scheduled earlier. This call is optional and is only issued if a sync point is desired prior to continued processing.
13. The program issues the EXEC CICS RETURN statement when it has finished its processing. If this is a return from the highest-level CICS program, a `TERM` call and sync point are internally generated by CICS.

Related reading: For more information about installing application programs, see *CICS Transaction Server for z/OS CICS Application Programming Guide*.

Related reference:

"Specifying the UIB (CICS online programs only)" on page 237

Coding a program in Java

IMS provides support for developing applications using the Java programming language.

You can write Java applications to access IMS databases and process IMS transactions by using the drivers and resource adapters of the IMS solutions for Java development.

Related concepts:

Chapter 35, "IMS solutions for Java development overview," on page 539

Coding a batch program in Pascal

The following code sample is a skeleton batch program in Pascal. It shows you how the parts of an IMS program that is written in Pascal fit together. The numbers to the right of the program refer to the notes that follow the program.

Restriction: Pascal is not supported by CICS.

```
segment PASCIMS;                                          NOTES
                                                             1
   type                                                      2
     CHAR2 = packed array [1..2] of CHAR;
```

```
  CHAR4 = packed array [1..4] of CHAR;
  CHAR6 = packed array [1..6] of CHAR;
  CHARn = packed array [1..n] of CHAR;
              DB_PCB_TYPE = record                          3
              DB_NAME : ALFA;
              DB_SEG_LEVEL : CHAR2;
              DB_STAT_CODE : CHAR2;
              DB_PROC_OPT  : CHAR4;
              FILLER       : INTEGER;
              DB_SEG_NAME  : ALFA;
              DB_LEN_KFB   : INTEGER;
              DB_NO_SENSEG : INTEGER;
              DB_KEY_FB    : CHARn;
            end;
procedure PASCIMS (var SAVE: INTEGER;                       4
                   var DB_PCB_MAST: DB_PCB_TYPE;
                    var DB_PCB_DETAIL : DB_PCB_TYPE);
              REENTRANT;
procedure PASCIMS;
type                                                        5
  QUAL_SSA_TYPE  = record
                             SEG_NAME           : ALFA;
                  SEQ_QUAL     : CHAR;
                  SEG_KEY_NAME : ALFA;
                  SEG_OPR      : CHAR2;
                  SEG_KEY_VALUE: CHAR6;
                  SEG_END_CHAR : CHAR;
                 end;
  MAST_SEG_IO_AREA_TYPE = record
                            (* Field declarations *)
                           end;
   DET_SEG_IO_AREA_TYPE = record
                            (* Field declarations *)
                          end;
var                                                         6
  MAST_SEG_IO_AREA : MAST_SEG_IO_AREA_TYPE;
  DET_SEG_IO_AREA : DET_SEG_IO_AREA_TYPE;
const                                                       7
   GU   = 'GU  ';
   GN   = 'GN  ';
  GHU  = 'GHU ';
  GHN  = 'GHN ';
  GHNP = 'GHNP';
  ISRT = 'ISRT';
  REPL = 'REPL';
  DLET = 'DLET';
  QUAL_SSA = QUAL_SSA_TYPE('ROOT','(','KEY',' =',
                           'vvvvv',')');
  UNQUAL_SSA = 'NAME     ';
 procedure PASTDLI; GENERIC;                                 8
begin
  PASTDLI(const GU,                                         9
          var   DB_PCB_DETAIL;
          var   DET_SEG_IO_AREA;
          const QUAL_SSA);
  PASTDLI(const GHU,                                       10
          var   DB_PCB_MAST,
          var   MAST_SEG_IO_AREA,
          const QUAL_SSA);
  PASTDLI(const GHN,                                       11
          var   DB_PCB_MAST,
          var   MAST_SEG_IO_AREA);
  PASTDLI(const REPL,                                      12
          var   DB_PCB_MAST,
          var   MAST_SEG_IO_AREA);
end;
                                                           13
```

Note:

1. Define the name of the Pascal compile unit.
2. Define the data types that are needed for the PCBs used in your program.
3. Define the PCB data type that is used in your program.
4. Declare the procedure heading for the REENTRANT procedure that is called by IMS. The first word in the parameter list should be an INTEGER, which is reserved for VS Pascal's usage. The rest of the parameters are the addresses of the PCBs that are received from IMS.
5. Define the data types that are needed for the SSAs and I/O areas.
6. Declare the variables used for the I/O areas.
7. Define the constants, such as function codes and SSAs that are used in the PASTDLI DL/I calls.
8. Declare the IMS interface routine by using the GENERIC directive. GENERIC identifies external routines that allow multiple parameter list formats. A GENERIC routine's parameters are “declared” only when the routine is called.
9. This call retrieves data from the database. It contains a qualified SSA. Before you can issue a call that uses a qualified SSA, you must initialize the data field of the SSA. Before you can issue a call that uses an unqualified SSA, you must initialize the segment name field.
10. This is another call that has a qualified SSA.
11. This call is an unqualified call that retrieves data from the database. Because it is a Get Hold call, it can be followed by a `REPL` or `DLET` call.
12. The `REPL` call replaces the data in the segment that was retrieved by the most recent Get Hold call; the data is replaced by the contents of the I/O area that is referenced in the call.
13. You return control to IMS by exiting from the PASCIMS procedure. You can also code a RETURN statement to exit at another point.

Restriction: You must bind your program to the IMS language interface module (DFSLI000) after compiling your program.

Coding a batch program in PL/I

The following code example is a skeleton batch program in PL/I. It shows you how the parts of an IMS program that is written in PL/I fit together.

The numbers to the right of the program refer to the notes that follow. This kind of program can run as a batch program or as a batch-oriented BMP.

Restriction: IMS application programs cannot use PL/I multitasking. This is because all tasks operate as subtasks of a PL/I control task when you use multitasking.

Sample PL/I program

```
/*                                                  */                  NOTES
/*               ENTRY POINT                        */
/*                                                  */
DLITPLI: PROCEDURE (IO_PTR_PCB,DB_PTR_MAST,DB_PTR_DETAIL)                  1
             OPTIONS (MAIN);
/*                                                  */
/*            DESCRIPTIVE STATEMENTS                */
/*                                                  */
DCL  IO_PTR_PCB POINTER;
DCL  DB_PTR_MAST POINTER;
DCL  DB_PTR_DETAIL POINTER;
```

```
DCL  FUNC_GU     CHAR(4)    INIT('GU  ');                                    2
DCL  FUNC_GN     CHAR(4)    INIT('GN  ');
DCL  FUNC_GHU    CHAR(4)    INIT('GHU ');
DCL  FUNC_GHN    CHAR(4)    INIT('GHN ');
DCL  FUNC_GNP    CHAR(4)    INIT('GNP ');
DCL  FUNC_GHNP   CHAR(4)    INIT('GHNP');
DCL  FUNC_ISRT   CHAR(4)    INIT('ISRT');
DCL  FUNC_REPL   CHAR(4)    INIT('REPL');
DCL  FUNC_DLET   CHAR(4)    INIT('DLET');
DCL 1    QUAL_SSA           STATIC UNALIGNED,                                3
         2   SEG_NAME       CHAR(8) INIT('ROOT    '),
         2   SEG_QUAL       CHAR(1) INIT('('),
         2   SEG_KEY_NAME   CHAR(8) INIT('KEY    '),
         2   SEG_OPR        CHAR(2) INIT(' ='),
         2   SEG_KEY_VALUE  CHAR(6) INIT('vvvvv'),
         2   SEG_END_CHAR   CHAR(1) INIT(')');
DCL 1    UNQUAL SSA         STATIC UNALIGNED,
         2   SEG_NAME_U     CHAR(8) INIT('NAME    '),
         2   BLANK          CHAR(1) INIT(' ');
DCL 1   MAST_SEG_IO_AREA,                                                    4
         2 ——
         2 ——
         2 ——
DCL 1   DET_SEG_IO_AREA,
         2 ——
         2 ——
         2 ——
DCL 1    IO_PCB               BASED (IO_PTR_PCB),                            5
         2   FILLER           CHAR(10),
         2   STAT             CHAR(2);
DCL 1    DB_PCB_MAST          BASED (DB_PTR_MAST),
         2   MAST_DB_NAME     CHAR(8),
         2   MAST_SEG_LEVEL   CHAR(2),
         2   MAST_STAT_CODE   CHAR(2),
         2   MAST_PROC_OPT    CHAR(4),
         2   FILLER           FIXED BINARY (31,0),
         2   MAST_SEG_NAME    CHAR(8),
         2   MAST_LEN_KFB     FIXED BINARY (31,0),
         2   MAST_NO_SENSEG   FIXED BINARY (31,0),
         2   MAST_KEY_FB      CHAR(*);
DCL 1    DB_PCB_DETAIL        BASE (DB_PTR_DETAIL),
         2   DET_DB_NAME      CHAR(8),
         2   DET_SEG_LEVEL    CHAR(2),
         2   DET_STAT_CODE    CHAR(2),
         2   DET_PROC_OPT     CHAR(4),
         2   FILLER           FIXED BINARY (31,0),
         2   DET_SEG_NAME     CHAR(8),
         2   DET_LEN_KFB      FIXED BINARY (31,0),
         2   DET_NO_SENSEG    FIXED BINARY (31,0),
         2   DET_KEY_FB       CHAR(*);
DCL      THREE    FIXED BINARY     (31,0)   INITIAL(3);                     6
DCL      FOUR     FIXED BINARY     (31,0)   INITIAL(4);
DCL      FIVE     FIXED BINARY     (31,0)   INITIAL(5);
DCL      SIX      FIXED BINARY     (31,0)   INITIAL(6);
/*                                                          */
/*            MAIN PART OF PL/I BATCH PROGRAM               */
/*                                                          */
CALL PLITDLI (FOUR,FUNC_GU,DB_PCB_DETAIL,DET_SEG_IO_AREA, QUAL_SSA);         7
   IF DET_STAT_CODE = GOOD_STATUS_CODE THEN DO;
      CALL PLITDLI (FOUR,FUNC_GHU,DB_PCB_MAST,MAST_SEG_IO_AREA,QUAL_SSA);    8
      IF MAST_STAT_CODE = GOOD_STATUS_CODE THEN DO;
         CALL PLITDLI (THREE,FUNC_GHN,DB_PCB_MAST,MAST_SEG_IO_AREA);         9
         IF MAST_STAT_CODE = GOOD_STATUS_CODE THEN DO;
            CALL PLITDLI (THREE,FUNC_REPL,DB_PCB_MAST,MAST_SEG_IO_AREA);     10
            IF MAST_STAT_CODE ^= GOOD_STATUS_CODE THEN DO;
               /* INSERT REPLACE DIAGNOSTIC MESSAGE */
          END;
```

```
   END;
         ELSE DO;
            /* INSERT GHN DIAGNOSTIC MESSAGE */
   END;
         END;
      ELSE DO;
         /* INSERT GHU DIAGNOSTIC MESSAGE */
        END;
END;
ELSE DO;
   /* INSERT GU DIAGNOSTIC MESSAGE */
END;
RETURN;                                                          11
END DLITPLI;
```

Note:

1. After IMS has loaded the PSB of the application program, IMS gives control to the application program through this entry point. PL/I programs must pass the pointers to the PCBs, not the names, in the entry statement. The entry statement lists the PCBs that the program uses by the names that it has assigned to the definitions for the PCB masks. The order in which you refer to the PCBs in the entry statement must be the same order in which they have been defined in the PSB.

   The code example assumes that an I/O PCB was passed to the application program. When the program is a batch program, CMPAT=YES must be specified on the PSBGEN statement of PSBGEN so that the I/O PCB is included. Because the I/O PCB is required for a batch program to make system service calls, CMPAT=YES should always be specified for batch programs.
2. Each of these areas defines one of the call functions used by the batch program. Each character string is defined as four alphanumeric characters, with a value assigned for each function. You can define other constants in the same way. Also, you can store standard definitions in a source library and include them by using a %INCLUDE statement.
3. A structure definition defines each SSA the program uses. The unaligned attribute is required for SSAs. The SSA character string must reside contiguously in storage. You should define a separate structure for each qualified SSA, because the value of the data field for each SSA is different.
4. The I/O areas that are used to pass segments to and from the database are defined as structures.
5. Level-01 declaratives define masks for the PCBs that the program uses as structures. These definitions make it possible for the program to check fields in the PCBs.
6. This statement defines the *parmcount* that is required in DL/I calls that are issued from PL/I programs (except for the call to the sample status-code error routine, where it is not allowed). The *parmcount* is the address of a 4-byte field that contains the number of subsequent parameters in the call. The *parmcount* is required only in PL/I programs. It is optional in the other languages. The value in *parmcount* is binary. This example shows how you can code the *parmcount* parameter when three parameters follow in the call:

   ```
   DCL    THREE    FIXED BINARY    (31,0)    INITIAL(3);
   ```
7. This call retrieves data from the database. It contains a qualified SSA. Before you can issue a call that uses a qualified SSA, initialize the data field of the SSA. Before you can issue a call that uses an unqualified SSA, initialize the segment name field. Check the status code after each DL/I call that you issue.

Although you must declare the PCB parameters that are listed in the entry statement to a PL/I program as POINTER data types, you can pass either the PCB name or the PCB pointer in DL/I calls in a PL/I program.

8. This is another call that has a qualified SSA.
9. This is an unqualified call that retrieves data from the database. Because it is a Get Hold call, it can be followed by `REPL` or `DLET`.
10. The `REPL` call replaces the data in the segment that was retrieved by the most recent Get Hold call; the data is replaced by the contents of the I/O area referenced in the call.
11. The RETURN statement returns control to IMS.

Binding PL/I code to the IMS language interface module

IMS provides a language interface module (DFSLI000) which gives a common interface to IMS. This module must be bound to the program.

If you use the IMS-supplied procedures (IMSPLI or IMSPLIGO), IMS binds the language interface module to the application program. IMSPLI is a two-step procedure that compiles and binds your program. IMSPLIGO is a three-step procedure that compiles, binds, and executes your program in a DL/I batch region. For information on how to use these procedures, see *IMS Version 11 System Definition*.

Coding a CICS online program in PL/I

The following code example is a skeleton CICS online program in PL/I. It shows you how to define and establish addressability to the UIB.

The numbers to the right of the program refer to the notes that follow. This kind of program can run in a CICS environment using DBCTL.

Sample call-level PL/I program (CICS online)

```
PLIUIB: PROC OPTIONS(MAIN);                                          NOTES
 DCL PSB_NAME CHAR(8) STATIC INIT('PLIPSB  ');                        1
 DCL PCB_FUNCTION CHAR(4) STATIC INIT('PCB ');
 DCL TERM_FUNCTION CHAR(4) STATIC INIT('TERM');
 DCL GHU_FUNCTION CHAR(4) STATIC INIT('GHU ');
 DCL REPL_FUNCTION CHAR(4) STATIC INIT('REPL');
 DCL SSA1 CHAR(9) STATIC INIT('AAAA4444 ');                           2
 DCL PARM_CT_1 FIXED BIN(31) STATIC INIT(1);
 DCL PARM_CT_3 FIXED BIN(31) STATIC INIT(3);
 DCL PARM_CT_4 FIXED BIN(31) STATIC INIT(4);
 DCL GOOD_RETURN_CODE BIT(8) STATIC INIT('0'B);
 DCL GOOD_STATUS_CODE CHAR(2) STATIC INIT('  ');
 %INCLUDE DLIUIB;                                                     3
 DCL 1 PCB_POINTERS BASED(UIBPCBAL),                                  4
       2 PCB1_PTR POINTER;
 DCL 1 DLI_IO_AREA,                                                   5
       2 AREA1 CHAR(3),
       2 AREA2 CHAR(37);
 DCL 1 PCB1 BASED(PCB1_PTR),                                          6
       2 PCB1_DBD_NAME CHAR(8),
       2 PCB1_SEG_LEVEL CHAR(2),
       2 PCB1_STATUS_CODE CHAR(2),
       2 PCB1_PROC_OPTIONS CHAR(4),
       2 PCB1_RESERVE_DLI FIXED BIN (31,0),
       2 PCB1_SEGNAME_FB CHAR(8),
       2 PCB1_LENGTH_FB_KEY FIXED BIN(31,0),
       2 PCB1_NUMB_SENS_SEGS FIXED BIN(31,0),
       2 PCB1_KEY_FB_AREA CHAR(17);
     /* SCHEDULE PSB AND OBTAIN PCB ADDRESSES */
```

```
CALL PLITDLI (PARM_CT_3,PCB_FUNCTION,PSB_NAME,UIBPTR);                    7
IF UIBFCTR = GOOD RETURN CODE THEN DO;
   /* ISSUE DL/I CALL: GET A UNIQUE SEGMENT */
   CALL PLITDLI (PARM_CT_4,GHU_FUNCTION,PCB1,DLI_IO_AREA,SSA1);           8
   IF UIBFCTR = GOOD_RETURN_CODE& PCB1_STATUS_CODE = GOOD_STATUS_CODE THEN DO; 9
      /* PERFORM SEGMENT UPDATE ACTIVITY */
     AREA1 = ......;
     AREA2 = ......;
    /* ISSUE DL/I: REPLACE SEGMENT AT CURRENT POSITION */
      PLITDLI (PARM_CT_3,REPL_FUNCTION,PCB1,DLI_IO_AREA);                 10
    IF UIBFCTR ^= GOOD_RETURN_CODE
     | PCB1_STATUS_CODE ^= GOOD_STATUS_CODE THEN DO;
     /* INSERT REPL ERROR DIAGNOSTIC CODE */
    END;
   END;
   ELSE DO;
      /* INSERT GHU ERROR DIAGNOSTIC CODE */
   END;
END;
ELSE DO;
   /* ANALYZE UIB PROBLEM */
   /* ISSUE UIB DIAGNOSTIC MESSAGE */
END;
/* RELEASE THE PSB */
CALL PLITDLI(PARM_CT_1,TERM_FUNCTION);                                    11
EXEC CICS RETURN;                                                         12
END PLIUIB;
```

Note:

1. Each of these areas defines the DL/I call functions the program uses. Each character string is defined as four alphanumeric characters and has a value assigned for each function. You can define other constants in the same way. You can store standard definitions in a source library and include them by using a %INCLUDE statement.
2. A structure definition defines each SSA the program uses. The unaligned attribute is required for SSA. The SSA character string must reside contiguously in storage. If a call requires two or more SSA, you may need to define additional areas.
3. The %INCLUDE DLIUIB statement will be expanded.
4. The UIB returns the address of an area containing the PCB addresses. The definition of PCB pointers is necessary to obtain the actual PCB addresses. Do not alter the addresses in the area.
5. The I/O areas that are used to pass segments to and from the database are defined as structures.
6. The PCBs are defined based on the addresses that are passed in the UIB.
7. The PCB call schedules a PSB for your program to use.
8. This unqualified GHU call retrieves a segment from the database. The segment is placed in the I/O area that is referenced in the call. Before issuing the call, the program must initialize the key or data value of the SSA so that it specifies the particular segment to be retrieved.
9. CICS online programs must test the return code in the UIB before testing the status code in the DB PCB.
10. The REPL call replaces the segment that was retrieved in the most recent Get Hold call. The I/O area that is referenced in the call contains the segment to be replaced.
11. The TERM call terminates the PSB that the program scheduled earlier.

12. The program issues the EXEC CICS RETURN statement when it has finished processing.

Related reading: For more information about installing application programs, see *CICS Transaction Server for z/OS CICS Application Programming Guide*.

Related reference:

"Specifying the UIB (CICS online programs only)" on page 237

"Coding a CICS online program in PL/I" on page 210

Chapter 12. Defining application program elements for IMS DB

Use these specific parameters and formats for making DL/I calls through the language interfaces for your applications program written in assembler language, C language, COBOL, Pascal, and PL/I.

Formatting DL/I calls for language interfaces

When you use DL/I calls in assembler language, C language, COBOL, Pascal, or PL/I, you must call the DL/I language interface to initiate the functions specified with the DL/I calls.

IMS offers several interfaces for DL/I calls:

- A language-independent interface for any programs that are Language Environment® conforming (CEETDLI)
- Language-specific interfaces for all supported languages (xxxTDLI)
- A non-language-specific interface for all supported languages (AIBTDLI)

Java makes use of the all three DL/I language interfaces, but the usage is internal and no calls are necessary to initiate the functions specified with the DL/I calls.

Related concepts:

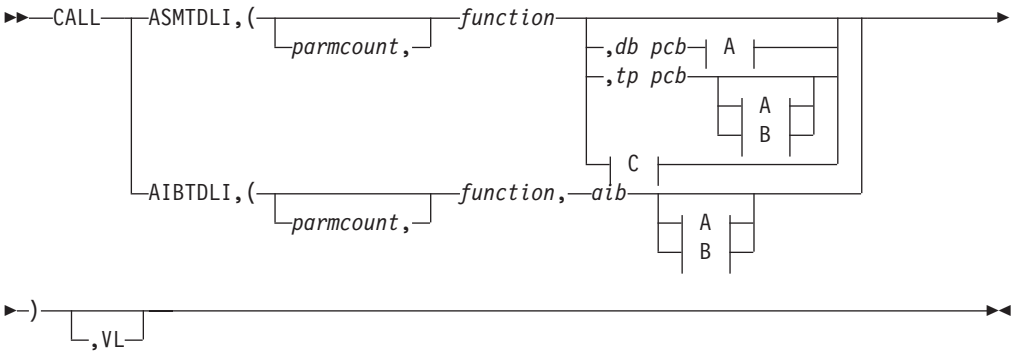
Chapter 35, "IMS solutions for Java development overview," on page 539

Assembler language application programming

Application programs in assembly language use the following format, parameters, and DL/I calls to communicate with IMS databases.

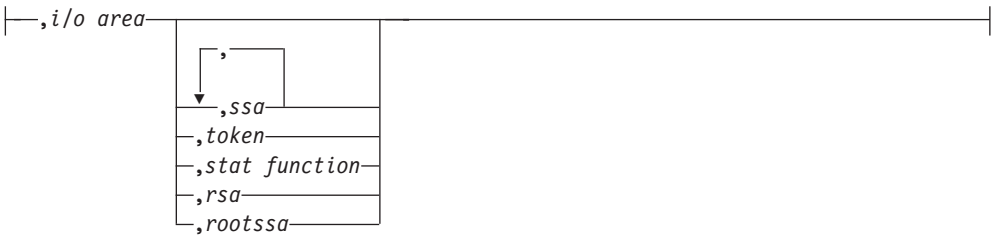
In assembler language programs, all DL/I call parameters that are passed as addresses can be passed in a register, which, if used, must be enclosed in parentheses.

Format

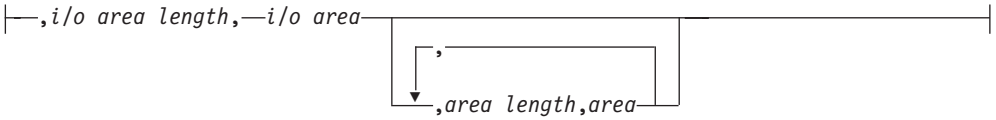


A:

© Copyright IBM Corp. 1974, 2015



B:



C:

Parameters

parmcount
Specifies the address of a 4-byte field in user-defined storage that contains the number of parameters in the parameter list that follows *parmcount*. Assembler language application programs must use either *parmcount* or **VL**.

function
Specifies the address of a 4-byte field in user-defined storage that contains the call function. The call function must be left-justified and padded with blanks (such as GU△△).

db pcb
Specifies the address of the database PCB to be used for the call. The PCB address must be one of the PCB addresses passed on entry to the application program in the PCB list.

tp pcb
Specifies the address of the I/O PCB or alternate PCB to be used for the call. The PCB address must be one of the PCB addresses passed on entry to the application program in the PCB list.

aib
Specifies the address of the application interface block (AIB) in user-defined storage.

i/o area
Specifies the address of the I/O area in user-defined storage that is used for the call. The I/O area must be large enough to contain the returned data.

i/o area length
Specifies the address of a 4-byte field in user-defined storage that contains the I/O area length (specified in binary).

area length
Specifies the address of a 4-byte field in user-defined storage that contains the

length (specified in binary) of the area immediately following it in the parameter list. Up to seven area lengths or area pairs can be specified.

area
Specifies the address of the area in user-defined storage to be checkpointed. Up to seven area lengths or area pairs can be specified.

token
Specifies the address of a 4-byte field in user-defined storage that contains a user token.

stat function
Specifies the address of a 9-byte field in user-defined storage that contains the stat function to be performed.

ssa
Specifies the address in user-defined storage that contains the SSAs to be used for the call. Up to 15 SSAs can be specified, one of which is *rootssa*.

rootssa
Specifies the address of a root segment search argument in user-defined storage.

rsa
Specifies the address of the area in user-defined storage that contains the record search argument.

psb name
Specifies the address in user-defined storage of an 8-byte PSB name to be used for the call.

uibptr
Specifies the address in user-defined storage of the user interface block (UIB).

sysserve
Specifies the address of an 8-byte field in user-defined storage to be used for the call.

VL
Signifies the end of the parameter list. Assembler language programs must use either *parmcount* or **VL**.

Example of a DL/I call format

Using the DL/I AIBTDLI interface:

```
CALL    AIBTDLI,(function,aib,i/o area,ssa1),VL
```

Using the DL/I language-specific interface:

```
CALL    ASMTDLI,(function,db pcb,i/o area,ssa1),VL
```

Related concepts:

"AIBTDLI interface" on page 245

Related reference:

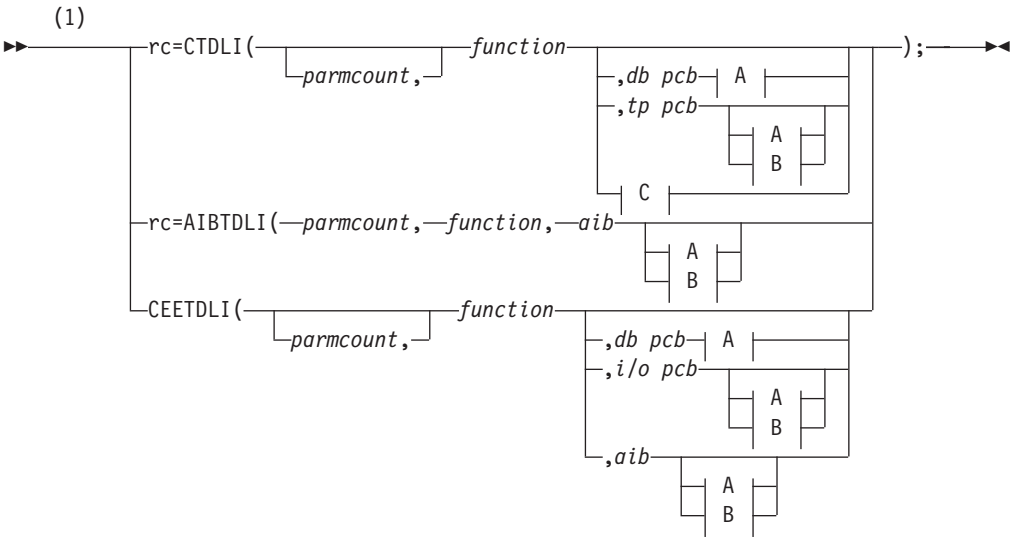
DL/I calls for database management (Application Programming APIs)

DL/I calls for IMS DB system services (Application Programming APIs)

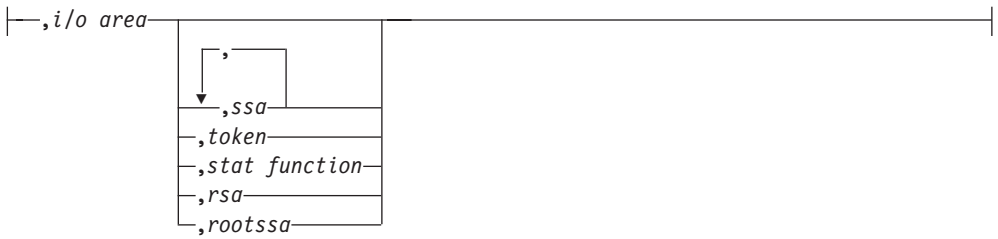
C language application programming

Application programs in C use the following format, parameters, and DL/I calls to communicate with IMS databases.

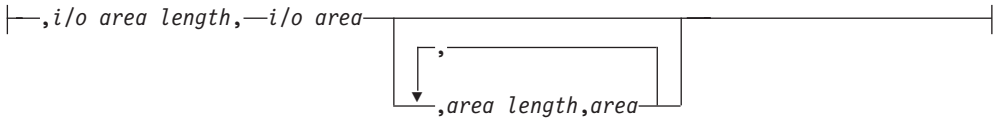
Format



A:



B:



C:

Notes:

1 For AIBTDLI, *parmcount* is required for C applications.

Parameters

rc This parameter receives the DL/I status or return code. It is a two-character field shifted into the 2 low-order bytes of an integer variable (int). If the status code is two blanks, 0 is placed in the field. You can test the **rc** parameter with an **if** statement. For example, `if (rc == 'IX')`. You can also use **rc** in a **switch** statement. You can choose to ignore the value placed in **rc** and use the status code returned in the PCB instead.

parmcount
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the number of parameters in the parameter list that follows parmcount.

function
Specifies the name of a character (4) variable, left justified in user-defined storage, that contains the call function to be used. The call function must be left-justified and padded with blanks (such as GU⌂

db pcb
Specifies the name of a pointer variable that contains the address of the database to be used for the call. The PCB address must be one of the PCB addresses passed on entry to the application program in the PCB list.

tp pcb
Specifies the name of a pointer variable that contains the address of the I/O PCB or alternate PCB to be used for the call. The PCB address must be one of the PCB addressed passed on entry to the application program in the PCB list.

aib
Specifies the name of the pointer variable that contains the address of the structure that defines the application interface block (AIB) in user-defined storage.

i/o area
Specifies the name of a pointer variable to a major structure, array, or character string that defines the I/O area in user-defined storage used for the call. The I/O area must be large enough to contain all of the returned data.

i/o area length
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the I/O area length.

area length
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the length of the area immediately following it in the parameter list. Up to seven area lengths or area pairs can be specified.

area
Specifies the name of the pointer variable that contains the address of the structure that defines the user-defined storage to be checkpointed. Up to seven area lengths or area pairs can be specified.

token
Specifies the name of a character (4) variable in user-defined storage that contains a user token.

stat function
Specifies the name of a character (9) variable in user-defined storage that contains the stat function to be performed.

ssa
Specifies the name of a character variable in user-defined storage that contains the SSAs to be used for the call. Up to 15 SSAs can be specified, one of which is *rootssa*.

rootssa
Specifies the name of a character variable that defines the root segment search argument in user-defined storage.

rsa
Specifies the name of a character variable that contains the record search argument for a GU call or where IMS should return the *rsa* for an ISRT or GN call.

psb name
Specifies the name of a character (8) variable containing the PSB name to be used for the call.

uibptr
Specifies the name of a pointer variable that contains the address of the structure that defines the user interface block (UIB) that is used in user-defined storage.

sysserve
Specifies the name of a character (8) variable string in user-defined storage to be used for the call.

I/O area

In C, the I/O area can be of any type, including structures or arrays. The **ctdli** declarations in **ims.h** do not have any prototype information, so no type checking of the parameters is done. The area may be **auto**, **static**, or allocated (with **malloc** or **calloc**). You need to give special consideration to C-strings because DL/I does not recognize the C convention of terminating strings with nulls ('\0') Instead of the usual **strcpy** and **strcmp** functions, you may want to use **memcpy** and **memcmp**.

Example of a DL/I call format

Using the DL/I CEETDLI interface:

```
#include <leawi.h>
⋮
CEETDLI (function,db pcb,i/o area,ssa1);
```

Using the DL/I AIBTDLI interface:

```
int rc;
⋮
rc=AIBTDLI (parmcount,function,aib,i/o area,ssa1);
```

Using the DL/I language-specific interface:

```
#include <ims.h>
int rc;
⋮
rc=CTDLI (function,db pcb,i/o area,ssa1);
```

Related concepts:

"AIBTDLI interface" on page 245

Related reference:

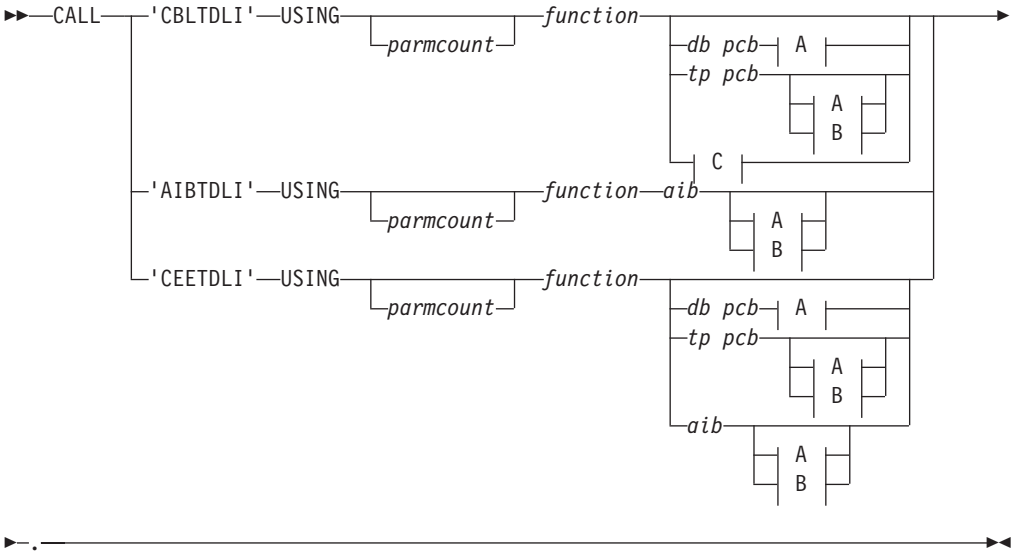
DL/I calls for database management (Application Programming APIs)

DL/I calls for IMS DB system services (Application Programming APIs)

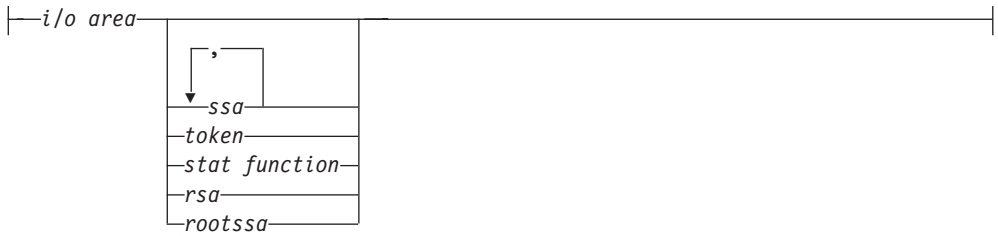
COBOL application programming

Application programs in COBOL use the following format, parameters, and DL/I calls to communicate with IMS databases.

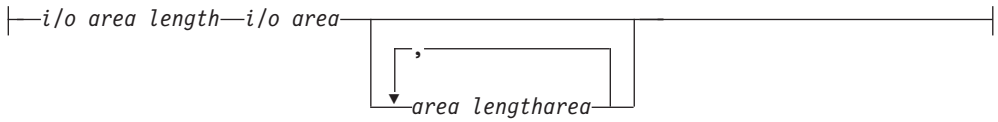
Format



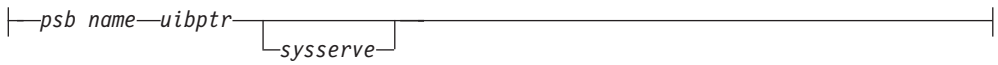
A:



B:



C:



Note: All apostrophes (') can be replaced by quotation marks (") and can be done regardless of the APOST/QUOTE compiler (or CICS translator) option.

Parameters

parmcount
: Specifies the identifier of a usage binary (4) byte data item in user-defined storage that contains the number of parameters in the parameter list that follows *parmcount*. If you define this field as COMP-5 rather than COMP,

COMP-4, or BINARY, then it can contain the maximum possible values regardless of the COBOL TRUNC compiler option setting.

function
Specifies the identifier of a usage display (4) byte data item, left justified in user-defined storage that contains the call function to be used. The call function must be left-justified and padded with blanks (such as GUbb).

db pcb
Specifies the identifier of the database PCB group item from the PCB list that is passed to the application program on entry. This identifier will be used for the call.

tp pcb
Specifies the identifier of the I/O PCB or alternate PCB group item from the PCB list that is passed to the application program on entry. This identifier will be used for the call.

aib
Specifies the identifier of the group item that defines the application interface block (AIB) in user-defined storage.

i/o area
Specifies the identifier of a major group item, table, or usage display data item that defines the I/O area length in user-defined storage used for the call. The I/O area must be large enough to contain all of the returned data.

i/o area length
Specifies the identifier of a usage binary (4) byte data item in user-defined storage that contains the I/O area length (specified in binary). If you define this field as COMP-5 rather than COMP, COMP-4, or BINARY, then it can contain the maximum possible values regardless of the COBOL TRUNC compiler option setting.

area length
Specifies the identifier of a usage binary (4) byte data item in user-defined storage that contains the length (specified in binary) of the area immediately following it in the parameter list. Up to seven area lengths or area pairs can be specified. If you define this field as COMP-5 rather than COMP, COMP-4, or BINARY, then it can contain the maximum possible values regardless of the COBOL TRUNC compiler option setting.

area
Specifies the identifier of the group item that defines the user-defined storage to be checkpointed. Up to seven area lengths or area pairs can be specified.

token
Specifies the identifier of a usage display (4) byte data item in user-defined storage that contains a user token.

stat function
Specifies the identifier of a usage display (9) byte data item in user-defined storage that contains the stat function to be performed.

ssa
Specifies the identifier of a usage display data item in user-defined storage that contains the SSAs to be used for the call. Up to 15 SSAs can be specified, one of which is *rootssa*.

rootssa
Specifies the identifier of a usage display data item that defines the root segment search argument in user-defined storage.

rsa
Specifies the identifier of a usage display data item that contains the record search argument.

psb name
Specifies the identifier of a usage display (8) byte data item containing the PSB name to be used for the call.

uibptr
Specifies the identifier of the group item that defines the user interface block (UIB) that is used in user-defined storage.

sysserve
Specifies the identifier of a usage display (8) byte data item in user-defined storage to be used for the call.

Example of a DL/I call format

Using the DL/I CEETDLI interface:

```
CALL    'CEETDLI' USING function,db pcb,i/o area,ssa1.
```

Using the DL/I AIBTDLI interface:

```
CALL    'AIBTDLI' USING function,aib,i/o area,ssa1.
```

Using the DL/I language-specific interface:

```
CALL    'CBLTDLI' USING function,db pcb,i/o area,ssa1.
```

Related reference:

DL/I calls for database management (Application Programming APIs)

DL/I calls for IMS DB system services (Application Programming APIs)

Java application programming for IMS

IMS provides support for developing applications using the Java programming language.

You can write Java applications to access IMS databases and process IMS transactions by using the drivers and resource adapters of the IMS solutions for Java development.

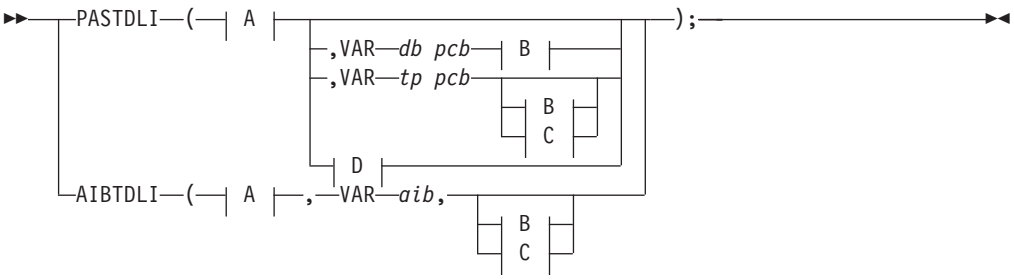
Related concepts:

Chapter 35, “IMS solutions for Java development overview,” on page 539

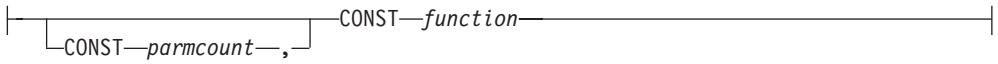
Pascal application programming

Application programs in Pascal use the following format, parameters, and DL/I calls to communicate with IMS databases.

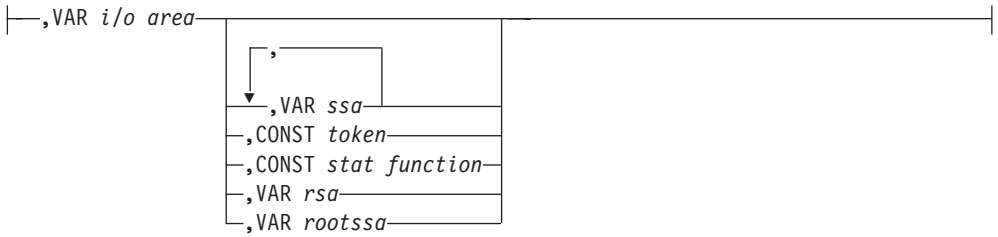
Format



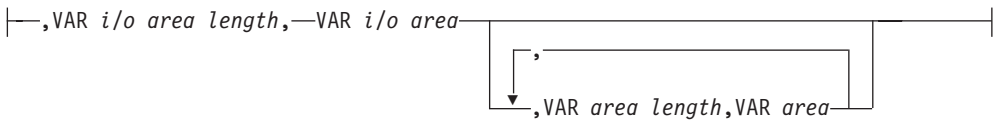
A:



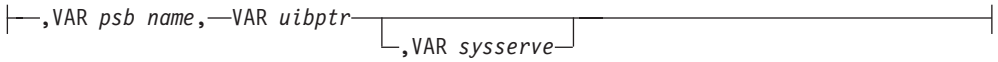
B:



C:



D:



Parameters

parmcount
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the number of parameters in the parameter list that follows parmcount.

function
Specifies the name of a character (4) variable, left justified in user-defined storage, that contains the call function to be used. The call function must be left-justified and padded with blanks (such as GU␣␣).

db pcb
Specifies the name of a pointer variable that contains the address of the database PCB defined in the call procedure statement.

tp pcb
Specifies the name of a pointer variable that contains the address of the I/O PCB or alternate PCB defined in the call procedure statement.

aib
Specifies the name of the pointer variable that contains the address of the structure that defines the application interface block (AIB) in user-defined storage.

i/o area
Specifies the name of a pointer variable to a major structure, array, or character string that defines the I/O area in user-defined storage used for the call. The I/O area must be large enough to contain all of the returned data.

i/o area length
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the I/O area length.

area length
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the length of the area immediately following it in the parameter list. Up to seven area lengths or area pairs can be specified.

area
Specifies the name of the pointer variable that contains the address of the structure that defines the user-defined storage to be checkpointed. Up to seven area lengths or area pairs can be specified.

token
Specifies the name of a character (4) variable in user-defined storage that contains a user token.

stat function
Specifies the name of a character (9) variable in user-defined storage that contains the stat function to be performed.

ssa
Specifies the name of a character variable in user-defined storage that contains the SSAs to be used for the call. Up to 15 SSAs can be specified, one of which is *rootssa*.

rootssa
Specifies the name of a character variable that defines the root segment search argument in user-defined storage.

rsa
Specifies the name of a character variable that contains the record search argument.

psb name
Specifies the name of a character (8) variable containing the PSB name to be used for the call.

uibptr
Specifies the name of a pointer variable that contains the address of the structure that defines the user interface block (UIB) that is used in user-defined storage.

sysserve
Specifies the name of a character (8) variable string in user-defined storage to be used for the call.

Example of a DL/I call format

Using the DL/I AIBTDLI interface:

```
AIBTDLI(CONST function,
  VAR aib,
  VAR i/o area,
  VAR ssa1);
```

Using the DL/I language-specific interface:

```
PASTDLI(CONST function,
  VAR db pcb,
  VAR i/o area,
  VAR ssa1);
```

Related reference:

DL/I calls for database management (Application Programming APIs)

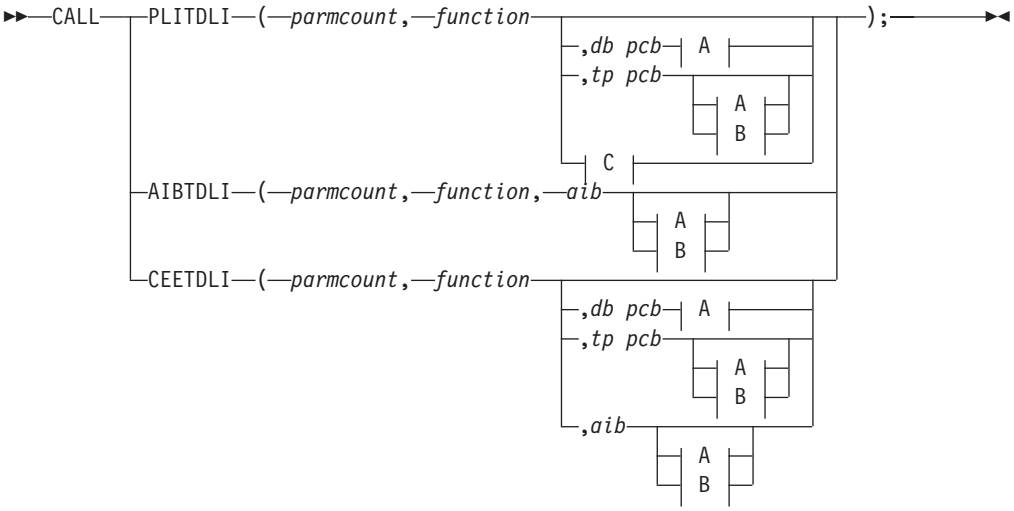
DL/I calls for IMS DB system services (Application Programming APIs)

Application programming for PL/I

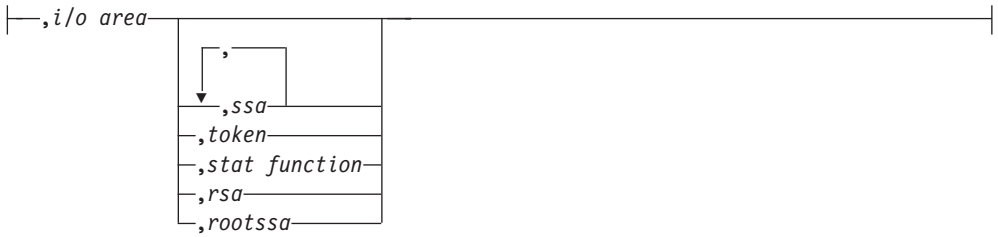
Application programs in PL/I use the following format, parameters, and DL/I calls to communicate with IMS databases.

Restriction: For the PLITDLI interface, all parameters except *parmcount* are indirect pointers; for the AIBTDLI interface, all parameters are direct pointers.

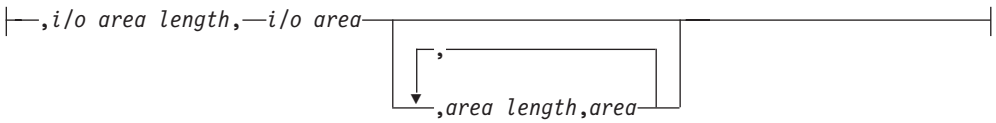
Format



A:



B:



C:

Parameters

parmcount
Specifies the name of a fixed binary (31-bit) variable that contains the number of arguments that follow *parmcount*.

function
Specifies the name of a fixed-character (4-byte) variable left-justified, blank padded character string containing the call function to be used (such as GU␣␣).

db pcb
Specifies the structure associated with the database PCB to be used for the call. This structure is based on a PCB address that must be one of the PCB addresses passed on entry to the application program.

tp pcb
Specifies the structure associated with the I/O PCB or alternate PCB to be used for the call.

aib
Specifies the name of the structure that defines the AIB in your application program.

i/o area
Specifies the name of the I/O area used for the call. The I/O area must be large enough to contain all the returned data.

i/o area length
Specifies the name of a fixed binary (31) variable that contains the I/O area length.

area length
Specifies the name of a fixed binary (31) variable that contains the length of the area immediately following it in the parameter list. Up to seven area lengths or area pairs can be specified.

area
Specifies the name of the area to be checkpointed. Up to seven area lengths or area pairs can be specified.

token
Specifies the name of a character (4) variable that contains a user token.

stat function
Specifies the name of a character (9) variable string containing the stat function to be performed.

ssa
Specifies the name of a character variable that contains the SSAs to be used for the call. Up to 15 SSAs can be specified, one of which is *rootssa*.

rootssa
Specifies the name of a character variable that contains a root segment search argument.

rsa
Specifies the name of a character variable that contains the record search argument.

psb name
Specifies the name of a character (8) containing the PSB name to be used for the call.

uibptr
Specifies the name of the user interface block (UIB).

sysserve
Specifies the name of a character (8) variable character string to be used for the call.

Example of a DL/I call format

Using the DL/I CEETDLI interface:

```
CALL    CEETDLI (parmcount,function,db pcb,i/o area,ssa1);
```

Using the DL/I AIBTDLI interface:

```
CALL    AIBTDLI (parmcount,function,aib,i/o area,ssa1);
```

Using the DL/I language-specific interface:

```
%INCLUDE CEEIBMAW;
CALL    PLITDLI (parmcount,function,db pcb,i/o area,ssa1);
```

Related reference:

DL/I calls for database management (Application Programming APIs)

DL/I calls for IMS DB system services (Application Programming APIs)

Specifying the I/O PCB mask

After your program issues a call with the I/O Program Communications Block (I/O PCB), IMS returns information about the results of the call to the I/O PCB. To determine the results of the call, your program must check the information that IMS returns.

Issuing a system service call requires an I/O PCB. Because the I/O PCB resides outside your program, you must define a mask of the PCB in your program to check the results of IMS calls. The mask must contain the same fields, in the same order, as the I/O PCB. Your program can then refer to the fields in the PCB through the PCB mask.

The following table shows the fields that the I/O PCB contains, their lengths, and the applicable environment for each field.

Table 34. I/O PCB mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Logical terminal name [1]	8	X		X		
Reserved for IMS [2]	2	X		X		
Status code [3]	2	X	X	X	X	X
4-Byte Local date and time [4]						
Date	4	X		X		
Time	4	X		X		
Input message sequence number [5]	4	X		X		
Message output descriptor name [6]	8	X		X		
Userid [7]	8	X		X		
Group name [8]	8	X		X		
12-Byte Time Stamp [9]						
Date	4	X		X		
Time	6	X		X		
UTC Offset	2	X		X		
Userid Indicator[10]	1	X		X		
Reserved for IMS[2]	3					

Note:

1. **Logical Terminal Name**

   This field contains the name of the terminal that sent the message. When your program retrieves an input message, IMS places the name of the logical terminal that sent the message in this field. When you want to send a message back to this terminal, you refer to the I/O PCB when you issue the `ISRT` call, and IMS takes the name of the logical terminal from the I/O PCB as the destination.

2. **Reserved for IMS**

   These fields are reserved.

3. **Status Code**

   IMS places the status code describing the result of the DL/I call in this field. IMS updates the status code after each DL/I call that the program issues. Your program should always test the status code after issuing a DL/I call.

   The three status code categories are:

   - Successful status codes or status codes with exceptional but valid conditions. This category does not contain errors. If the call was completely successful, this field contains blanks. Many of the codes in this category are for information only. For example, a QC status code means that no more messages exist in the message queue for the program. When your program receives this status code, it should terminate.

- Programming errors. The errors in this category are usually ones that you can correct. For example, an AD status code indicates an invalid function code.
- I/O or system errors.

For the second and third categories, your program should have an error routine that prints information about the last call that was issued before program termination. Most installations have a standard error routine that all application programs at the installation use.

4. **Local Date and Time**

   The current local date and time are in the prefix of all input messages except those originating from non-message-driven BMPs. The local date is a packed-decimal, right-aligned date, in the format `yyddd`. The local time is a packed-decimal time in the format `hhmmsst`. The current local date and time indicate when IMS received the entire message and enqueued it as input for the program, rather than the time that the application program received the message. To obtain the application processing time, you must use the time facility of the programming language you are using.

   For a conversation, for an input message originating from a program, or for a message received using Multiple System Coupling (MSC), the time and date indicate when the original message was received from the terminal.

5. **Input Message Sequence Number**

   The input message sequence number is in the prefix of all input messages except those originating from non-message-driven BMPs. This field contains the sequence number IMS assigned to the input message. The number is binary. IMS assigns sequence numbers by physical terminal, which are continuous since the time of the most recent IMS startup.

6. **Message Output Descriptor Name**

   You only use this field when you use MFS. When you issue a `GU` call with a message output descriptor (MOD), IMS places its name in this area. If your program encounters an error, it can change the format of the screen and send an error message to the terminal by using this field. To do this, the program must change the MOD name by including the MOD name parameter on an `ISRT` or `PURG` call.

   Although MFS does not support APPC, LU 6.2 programs can use an interface to emulate MFS. For example, the application program can use the MOD name to communicate with IMS to specify how an error message is to be formatted.

   Related reading: For more information on the MOD name and the LTERM interface, see *IMS Version 11 Communications and Connections*.

7. **Userid**

   The use of this field is connected with RACF signon security. If signon is not active in the system, this field contains blanks.

   If signon is active in the system, the field contains one of the following:

   - The user's identification from the source terminal.
   - The LTERM name of the source terminal if signon is not active for that terminal.
   - The authorization ID. For batch-oriented BMPs, the authorization ID is dependent on the value specified for the BMPUSID= keyword in the DFSDCxxx PROCLIB member:
     - If BMPUSID=USERID is specified, the value from the USER= keyword on the JOB statement is used.

- If USER= is not specified on the JOB statement, the program's PSB name is used.
- If BMPUSID=PSBNAME is specified, or if BMPUSID= is not specified at all, the program's PSB name is used.

8. **Group Name**

   The group name, which is used by DB2 to provide security for SQL calls, is created through IMS transactions.

   Three instances that apply to the group name are:

   - If you use RACF and SIGNON on your IMS system, the RACROUTE SAF (extract) call returns an eight-character group name.
   - If you use your own security package on your IMS system, the RACROUTE SAF call returns any eight-character name from the package and treats it as a group name. If the RACROUTE SAF call returns a return code of 4 or 8, a group name was not returned, and IMS blanks out the group name field.
   - If you use LU 6.2, the transaction header can contain a group name.

     Related reading: For more information about LU 6.2, see *IMS Version 11 Communications and Connections*.

9. **12-Byte Time Stamp**

   This field contains the current date and time fields, but in the IMS internal packed-decimal format. The time stamp has the following parts:

   Date `yyyydddf`

   This packed-decimal date contains the year (`yyyy`), day of the year (`ddd`), and a valid packed-decimal + sign such as (`f`).

   Time `hhmmssthmiju`

   This packed-decimal time consists of hours, minutes, and seconds (`hhmmss`) and fractions of the second to the microsecond (`thmiju`). **No** packed-decimal sign is affixed to this part of the time stamp.

   UTC Offset
   `aqq$`

   The packed-decimal UTC offset is prefixed by 4 bits of attributes (`a`). If the 4th bit of (`a`) is 0, the time stamp is UTC; otherwise, the time stamp is local time. The control region parameter, `TSR=(U/L)`, specified in the DFSPBxxx PROCLIB member, controls the representation of the time stamp with respect to local time versus UTC time.

   The offset value (`qq$`) is the number of quarter hours of offset to be added to UTC or local time to convert to local or UTC time respectively.

   The offset sign (`$`) follows the convention for a packed-decimal plus or minus sign.

   Field 4 always contains the local date and time.

   Related reading: For a more detailed description of the internal packed-decimal time-format, see *IMS Version 11 Operations and Automation*.

10. **Userid Indicator**

    The Userid Indicator is provided in the I/O PCB and in the response to the INQY call. The Userid Indicator contains one of the following:

    - U - The user's identification from the source terminal during signon
    - L - The LTERM name of the source terminal if signon is not active

- P - The PSBNAME of the source BMP or transaction
- O - Other name

The value contained in the Userid Indicator field indicates the contents of the userid field.

Specifying the DB PCB mask

IMS describes the results of the calls your program issues in the DB PCB that is referenced in the call. To determine the success or failure of the DL/I call, the application program includes a mask of the DB PCB and then references the fields of the DB PCB through the mask.

A DB PCB mask must contain the fields shown in the following table. (Your program can look at, but not change, the fields in the DB PCB.) The fields in your DB PCB mask must be defined in the same order and with the same length as the fields shown here. When you code the DB PCB mask, you also give it a name, but the name is not part of the mask. You use the name (or the pointer, for PL/I) when you reference each of the PCBs your program processes. A GSAM DB PCB mask is slightly different from other DB PCB masks.

Of the nine fields, only five are important to you as you construct the program. These are the fields that give information about the results of the call. They are the segment level number, status code, segment name, length of the key feedback area, and key feedback area. The status code is the field your program uses most often to find out whether the call was successful. The key feedback area contains the data from the segments you have specified; the level number and segment name help you determine the segment type you retrieved after an unqualified `GN` or `GNP` call, or they help you determine your position in the database after an error or unsuccessful call.

Table 35. DB PCB mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Database name [1]	8	X	X		X	
Segment level number [2]	2	X	X		X	
Status code [3]	2	X	X		X	
Processing options [4]	4	X	X		X	
Reserved for IMS [5]	4	X	X		X	
Segment name [6]	8	X	X		X	
Length of key feedback area [7]	4	X	X		X	
Number of sensitive segments [8]	4	X	X		X	
Key feedback area [9]	var length	X	X		X	

Note:

1. This contains the name of the database. This field is 8 bytes long and contains character data.
2. **Segment Level Number**

This field contains numeric character data. It is 2 bytes long and right-justified. When IMS retrieves the segment you have requested, IMS places the level number of that segment in this field. If you are retrieving several segments in a hierarchic path with one call, IMS places the number of the lowest-level segment retrieved. If IMS is unable to find the segment that you request, it gives you the level number of the last segment it encounters that satisfied your call.

3. **Status Code**

   After each DL/I call, this field contains the two-character status code that describes the results of the DL/I call. IMS updates this field after each call and does not clear it between calls. The application program should test this field after each call to find out whether the call was successful.

   When the program is initially scheduled, this field contains a data-availability status code, which indicates any possible access constraint based on segment sensitivity and processing options.

   Related Reading: For more information on these status codes, see the topic "INIT Call" in *IMS Version 11 Application Programming APIs*.

   During normal processing, four categories of status codes exist:

   - Successful or exceptional but valid conditions. If the call was completely successful, this field contains blanks. Many of the codes in this category are for information only. For example, GB means that IMS has reached the end of the database without satisfying the call. This situation is expected in sequential processing and is not usually the result of an error.
   - Errors in the program. For example, AK means that you have included an invalid field name in a segment search argument (SSA). Your program should have error routines available for these status codes. If IMS returns an error status code to your program, your program should terminate. You can then find the problem, correct it, and restart your program.
   - I/O or system error. For example, an AO status code means that there has been an I/O error concerning OSAM, BSAM, or VSAM. If your program encounters a status code in this category, it should terminate immediately. This type of error cannot normally be fixed without a system programmer, database administrator, or system administrator.
   - Data-availability status codes. These are returned only if your program has issued the `INIT` call indicating that it is prepared to handle such status codes. “Status Code Explanations” in *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes* describes possible causes and corrections in more detail.

4. **Processing Options**

   This is a 4-byte field containing a code that tells IMS what type of calls this program can issue. It is a security mechanism in that it can prevent a particular program from updating the database, even though the program can read the database. This value is coded in the PROCOPT parameter of the PCB statement when the PSB for the application program is generated. The value does not change.

5. **Reserved for IMS**

   This 4-byte field is used by IMS for internal linkage. It is not used by the application program.

6. **Segment Name**

   After each successful call, IMS places in this field the name of the last segment that satisfied the call. When a retrieval is successful, this field contains the name of the retrieved segment. When a retrieval is unsuccessful, this field

contains the last segment along the path to the requested segment that would satisfy the call. The segment name field is 8 bytes long.

When a program is initially scheduled, the name of the database type is put in the SEGNAME field. For example, the field contains DEDB when the database type is DEDB; GSAM when the database type is GSAM; HDAM, or PHDAM when the database type is HDAM or PHDAM.

7. **Length of Key Feedback Area**

   This is a 4-byte binary field that gives the current length of the key feedback area. Because the key feedback area is not usually cleared between calls, the program needs to use this length to determine the length of the relevant current concatenated key in the key feedback area.

8. **Number of Sensitive Segments**

   This is a 4-byte binary field that contains the number of segment types in the database to which the application program is sensitive.

9. **Key Feedback Area**

   At the completion of a retrieval or `ISRT` call, IMS places the concatenated key of the retrieved segment in this field. The length of the key for this request is given in the 4-byte field. If IMS is unable to satisfy the call, the key feedback area contains the key of the segment at the last level that was satisfied. A segment's concatenated key is made up of the keys of each of its parents and its own key. Keys are positioned left to right, starting with the key of the root segment and following the hierarchic path. IMS does not normally clear the key feedback area. IMS sets this length of the key feedback area to indicate the portion of the area that is valid at the completion of each call. Your program should not use the content of the key feedback area that is not included in the key feedback area length.

Related concepts:

"Data areas in GSAM databases" on page 245

Specifying the AIB mask

The application interface block (AIB) is used by your program to communicate with IMS, when your application does not have a PCB address or the call function does not use a PCB.

The application program can use the returned PCB address, when available, to inspect the status code in the PCB and to obtain any other information needed by the application program. The AIB mask enables your program to interpret the control block defined. The AIB structure must be defined in working storage, on a fullword boundary, and initialized according to the order and byte length of the fields as shown in the following table. The table's notes describe the contents of each field.

Table 36. AIB fields

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
AIB identifier	8	X	X	X	X	X
DFSAIB allocated length	4	X	X	X	X	X
Subfunction code	8	X	X	X	X	X
Resource name	8	X	X	X	X	X
Reserved 1	16					

Table 36. AIB fields (continued)

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Maximum output area length	4	X	X	X	X	X
Output area length used	4	X	X	X	X	X
Reserved 2	12					
Return code	4	X	X	X	X	X
Reason code	4	X	X	X	X	X
Error code extension	4	X		X		
Resource address	4	X	X	X	X	X
Reserved 3	48					

AIB Identifier (AIBID)

This 8-byte field contains the AIB identifier. You must initialize AIBID in your application program to the value DFSAIB ␣␣ before you issue DL/I calls. This field is required. When the call is completed, the information returned in this field is unchanged.

DFSAIB Allocated Length (AIBLEN)

This field contains the actual 4-byte length of the AIB as defined by your program. You must initialize AIBLEN in your application program before you issue DL/I calls. The minimum length required is 128 bytes. When the call is completed, the information returned in this field is unchanged. This field is required.

Subfunction Code (AIBSFUNC)

This 8-byte field contains the subfunction code for those calls that use a subfunction. You must initialize AIBSFUNC in your application program before you issue DL/I calls. When the call is completed, the information returned in this field is unchanged.

Resource Name (AIBRSNM1)

This 8-byte field contains the name of a resource. The resource varies depending on the call. You must initialize AIBRSNM1 in your application program before you issue DL/I calls. When the call is complete, the information returned in this field is unchanged. This field is required.

For PCB related calls where the AIB is used to pass the PCB name instead of passing the PCB address in the call list, this field contains the PCB name. The PCB name for the I/O PCB is IOPCB␣␣ The PCB name for other types of PCBs is defined in the PCBNAME= parameter in PSBGEN.

Reserved 1

This 16-byte field is reserved.

Maximum Output Area Length (AIBOALEN)

This 4-byte field contains the length of the output area in bytes that was specified in the call list. You must initialize AIBOALEN in your application program for all calls that return data to the output area. When the call is completed, the information returned in this area is unchanged.

Used Output Area Length (AIBOAUSE)

This 4-byte field contains the length of the data returned by IMS for all calls that return data to the output area. When the call is completed this field contains the length of the I/O area used for this call.

Reserved 2
This 12-byte field is reserved.

Return code (AIBRETRN)
When the call is completed, this 4-byte field contains the return code.

Reason Code (AIBREASN)
When the call is completed, this 4-byte field contains the reason code.

Error Code Extension (AIBERRXT)
This 4-byte field contains additional error information depending on the return code in AIBRETRN and the reason code in AIBREASN.

Resource Address (AIBRSA1)
When the call is completed, this 4-byte field contains call-specific information. For PCB related calls where the AIB is used to pass the PCB name instead of passing the PCB address in the call list, this field returns the PCB address.

Reserved 3

This 48-byte field is reserved.

Specifying the AIB mask for ODBA applications

The following table describes the fields for specifying the application interface block (AIB) mask for ODBA applications.

The notes that follow describe the contents of each field.

Table 37. AIB fields for use of ODBA applications

AIB Fields	Byte Length	DB/DC	IMS DB	DCCTL	DB Batch	TM Batch
AIB identifier	8	X	X	X	X	X
DFSAIB allocated length	4	X	X	X	X	X
Subfunction code	8	X	X	X	X	X
Resource name #1	8	X	X	X	X	X
Resource name #2	8					
Reserved 1	8	X				
Maximum output area length	4	X	X	X	X	X
Output area length used	4	X	X	X	X	X
Reserved 2	12					
Return code	4	X	X	X	X	X
Reason code	4	X	X	X	X	X
Error code extension	4	X				
Resource address #1	4	X	X	X	X	X
Resource address #2	4					
Resource address #3	4					
Reserved 3	40					
Reserved for ODBA	136					

AIB Identifier (AIBID)
This 8-byte field contains the AIB identifier. You must initialize AIBID in your application program to the value DFSAIBbb before you issue DL/I calls. This field is required. When the call is completed, the information returned in this field is unchanged.

DFSAIB Allocated Length (AIBLEN)
This field contains the actual 4-byte length of the AIB as defined by your program. You must initialize AIBLEN in your application program before you issue DL/I calls. The minimum length required is 264 bytes for ODBA. When the call is completed, the information returned in this field is unchanged. This field is required.

Subfunction Code (AIBSFUNC)
This 8-byte field contains the subfunction code for those calls that use a subfunction. You must initialize AIBSFUNC in your application program before you issue DL/I calls. When the call is completed, the information returned in this field is unchanged.

Resource Name (AIBRSNM1) #1

This 8-byte field contains the name of a resource. The resource varies depending on the call. You must initialize AIBRSNM1 in your application program before you issue DL/I calls. When the call is complete, the information returned in this field is unchanged. This field is required.

For PCB related calls where the AIB is used to pass the PCB name instead of passing the PCB address in the call list, this field contains the PCB name. The PCB name for the I/O PCB is IOPCBbb. The PCB name for other types of PCBs is defined in the PCBNAME= parameter in PSBGEN.

Resource Name (AIBRSNM2) #2
Specify a 4-character ID of ODBA startup table DFSxxxx0, where xxxx is a four-character ID.

Reserved 1
This 8-byte field is reserved.

Maximum Output Area Length (AIBOALEN)
This 4-byte field contains the length of the output area in bytes that was specified in the call list. You must initialize AIBOALEN in your application program for all calls that return data to the output area. When the call is completed, the information returned in this area is unchanged.

Used Output Area Length (AIBOAUSE)
This 4-byte field contains the length of the data returned by IMS for all calls that return data to the output area. When the call is completed this field contains the length of the I/O area used for this call.

Reserved 2
This 12-byte field is reserved.

Return code (AIBRETRN)
When the call is completed, this 4-byte field contains the return code.

Reason Code (AIBREASN)
When the call is completed, this 4-byte field contains the reason code.

Error Code Extension (AIBERRXT)
This 4-byte field contains additional error information depending on the return code in AIBRETRN and the reason code in AIBREASN.

Resource Address (AIBRSA1) #1
When the call is completed, this 4-byte field contains call-specific information. For PCB related calls where the AIB is used to pass the PCB name instead of passing the PCB address in the call list, this field returns the PCB address.

Resource Address (AIBRSA2) #2
This 4-byte field is reserved for ODBA.

Resource Address (AIBRSA3) #3
This 4-byte token, returned on the APSB call, is required for subsequent DLI calls and the DPSB call related to this thread.

Reserved 3

This 40-byte field is reserved.

Reserved for ODBA
This 136-byte field is reserved for ODBA.

The application program can use the returned PCB address, when available, to inspect the status code in the PCB and to obtain any other information needed by the application program.

COBOL AIB Mask Example

```
01  AIB.
    02 AIBRID            PIC x(8).
    02 AIBRLEN           PIC 9(9) USAGE BINARY.
    02 AIBRSFUNC         PIC x(8).
    02 AIBRSNM1          PIC x(8).
    02 AIBRSNM2          PIC x(8).
    02 AIBRESV1          PIC x(8).
    02 AIBOALEN          PIC 9(9) USAGE BINARY.
    02 AIBOAUSE          PIC 9(9) USAGE BINARY.
    02 AIBRESV2          PIC x(12).
    02 AIBRETRN          PIC 9(9) USAGE BINARY.
    02 AIBREASN          PIC 9(9) USAGE BINARY.
    02 AIBERRXT          PIC 9(9) USAGE BINARY.
    02 AIBRESA1          USAGE POINTER.
    02 AIBRESA2          USAGE POINTER.
    02 AIBRESA3          USAGE POINTER.
    02 AIBRESV4          PIC x(40).
    02 AIBRSAVE     OCCURS 18 TIMES USAGE POINTER.
    02 AIBRTOKN     OCCURS 6 TIMES  USAGE POINTER.
    02 AIBRTOKC          PIC x(16).
    02 AIBRTOKV          PIC x(16).
    02 AIBRTOKA     OCCURS 2 TIMES PIC 9(9) USAGE BINARY.
```

Assembler AIB Mask Example

```
DFSAIB   DSECT
AIBID    DS    CL8'DFSAIB'
AIBLEN   DS    F
AIBSFUNC DS    CL8
AIBRSNM1 DS    CL8
AIBRSVM2 DS    CL8
         DS    2F
AIBOALEN DS    F
AIBOAUSE DS    F
         DS    2F
         DS    H
         DS    H
AIBRETRN DS    F
AIBREASN DS    F
AIBRRXT  DS    F
```

```
AIBRSA1  DS    A
AIBRSA2  DS    A
AIBRSA3  DS    A
         DS    10F
AIBLL    EQU   *-DFSAIB
AIBSAVE  DS    18F
AIBTOKN  DS    6F
AIBTOKC  DS    CL16
AIBTOKV  DS    XL16
AIBTOKA  DS    2F
AIBAERL  EQU   *-DFSAIB
```

Specifying the UIB (CICS online programs only)

The interface between your CICS online program and DL/I passes additional information to your program in a user interface block (UIB). The UIB contains the address of the PCB list and any return codes your program must examine before checking the status code in the DB PCB.

When you issue the PCB call to obtain a PSB for your program, a UIB is created for your program. As with any area outside your program, you must include a definition of the UIB and establish addressability to it. CICS provides a definition of the UIB for all programming languages:

- In COBOL programs, use the COPY DLIUIB statement.
- In PL/I programs, use a %INCLUDE DLIUIB statement.
- In assembler language programs, use the DLIUIB macro.

Three fields in the UIB are important to your program: UIBPCBAL, UIBFCTR, and UIBDLTR. UIBPCBAL contains the address of the PCB address list. Through it you can obtain the address of the PCB you want to use. Your program must check the return code in UIBFCTR (and possibly UIBDLTR) before checking the status code in the DB PCB. If the contents of UIBFCTR and UIBDLTR are not null, the content of the status code field in the DB PCB is not meaningful. The return codes are described in the topic "CICS-DL/I user interface block return codes" in *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes*.

Immediately after the statement that defines the UIB in your program, you must define the PCB address list and the PCB mask.

The following code example shows how to use the COPY DLIUIB statement in a VS COBOL II program:

Defining the UIB, PCB address list, and the PCB mask for VS COBOL II

```
LINKAGE SECTION.

    COPY DLIUIB.
01  OVERLAY-DLIUIB REDEFINES DLIUIB.
    02  PCBADDR USAGE IS POINTER.
    02  FILLER PIC XX.

01  PCB-ADDRESSES.
    02  PCB-ADDRESS-LIST
            USAGE IS POINTER OCCURS 10 TIMES.
01  PCB1.
    02  PCB1-DBD-NAME PIC X(8).
```

```
    02  PCB1-SEG-LEVEL PIC XX.
        .
        .
        .
```

The COBOL COPY DLIUIB copybook

```
01  DLIUIB.
*                                   Address of the PCB addr list
    02  UIBPCBAL PIC S9(8) COMP.
*                                   DL/I return codes
    02  UIBRCODE.
*                                   Return codes
        03  UIBFCTR  PIC X.
            88  FCNORESP     VALUE ' '.
            88  FCNOTOPEN    VALUE ' '.
            88  FCINVREQ     VALUE ' '.
            88  FCINVPCB     VALUE ' '.
*                                   Additional information
        03  UIBDLTR  PIC X.
            88  DLPSBNF      VALUE ' '.
            88  DLTASKNA     VALUE ' '.
            88  DLPSBSCH     VALUE ' '.
            88  DLLANGCON    VALUE ' '.
            88  DLPSBFAIL    VALUE ' '.
            88  DLPSBNA      VALUE ' '.
            88  DLTERMNS     VALUE ' '.
            88  DLFUNCNS     VALUE ' '.
            88  DLINA        VALUE ' '.
```

The values placed in level 88 entries are not printable. They are described in the topic "CICS-DL/I User Interface Block Return Codes" in *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes*. The meanings of the field names and their hexadecimal values are shown below:

FCNORESP
Normal response Value X'00'

FCNOTOPEN
Not open Value X'0C'

FCINVREQ
Invalid request Value X'08'

FCINVPCB
Invalid PCB Value X'10'

DLPSBNF
PSB not found Value X'01'

DLTASKNA
Task not authorized Value X'02'

DLPSBSCH
PSB already scheduled Value X'03'

DLLANGCON
Language conflict Value X'04'

DLPSBFAIL
PSB initialization failed Value X'05'

DLPSBNA
PSB not authorized Value X'06'

DLTERMNS
Termination not successful Value X'07'

DLFUNCNS
Function unscheduled Value X'08'

DLINA
DL/I not active Value X'FF'

The following code example shows how to define the UIB, PCB address list, and PCB mask for PL/I.

Defining the UIB, PCB address list, and the PCB mask for PL/I

```
DCL UIBPTR PTR;                     /* POINTER TO UIB             */
DCL 1 DLIUIB UNALIGNED BASED(UIBPTR),
                                    /* EXTENDED CALL USER INTFC BLK*/
  2 UIBPCBAL PTR,                   /* PCB ADDRESS LIST           */
  2 UIBRCODE,                       /* DL/I RETURN CODES          */
    3 UIBFCTR BIT(8) ALIGNED,       /* RETURN CODES               */
    3 UIBDLTR BIT(8) ALIGNED;       /* ADDITIONAL INFORMATION     */
```

The following code example shows how to define the UIB, PCB address list, and PCB mask for assembler language.

Defining the UIB, PCB address list, and the PCB mask for assembler language

```
DLIUIB   DSECT
UIB      DS    0F                 EXTENDED CALL USER INTFC BLK
UIBPCBAL DS    A                  PCB ADDRESS LIST
UIBRCODE DS    0XL2               DL/I RETURN CODES
UIBFCTR  DS    X                  RETURN CODE
UIBDLTR  DS    X                  ADDITIONAL INFORMATION
         DS    2X                 RESERVED
         DS    0F                 LENGTH IS FULLWORD MULTIPLE
UIBLEN   EQU   *-UIB              LENGTH OF UIB
```

Related reference:

"Coding a CICS online program in COBOL" on page 201

"Coding a CICS online program in PL/I" on page 210

"Coding a CICS online program in assembler language" on page 194

"Language specific entry points" on page 246

Specifying the I/O areas

Use an I/O area to pass segments between the application program and IMS.

What the I/O area contains depends on the type of call you are issuing:

- When you retrieve a segment, IMS places the segment you requested in the I/O area.
- When you add a new segment, you first build the new segment in the I/O area.
- Before modifying a segment, your program must first retrieve it. When you retrieve the segment, IMS places the segment in an I/O area.

The format of the record segments you pass between your program and IMS can be fixed length or variable length. Only one difference is important to the application program: a message segment containing a 2-byte length field (or 4 bytes for the PLITDLI interface) at the beginning of the data area of the segment.

The I/O area for IMS calls must be large enough to hold the largest segment your program retrieves from or sends to IMS.

If your program issues any Get or ISRT calls that use the D command code, the I/O area must be large enough to hold the largest path of segments that the program retrieves or inserts.

Formatting segment search arguments (SSAs)

Segment search arguments in your assembler language, C language, COBOL, Java, Pascal, and PL/I application programs must be coded according to the following rules and formats.

SSA coding rules

Use the following rules for coding a segment search argument.

- Define the SSA in the data area of your program.
- The segment name field must:
  - Be 8 bytes long. If the name of the segment you are specifying is less than 8 bytes long, it should be left justified and padded on the right with blanks.
  - Contain a segment name that has been defined in the DBD that your application program uses. In other words, make sure you use the exact segment name, or your SSA will be invalid.
- If the SSA contains only the segment name, byte 9 must contain a blank.
- If the SSA contains one or more command codes:
  - Byte 9 must contain an asterisk (*).
  - The last command code must be followed by a blank unless the SSA contains a qualification statement. If the SSA contains a qualification statement, the command code must be followed by the left parenthesis of the qualification statement.
- If the SSA contains a qualification statement:
  - The qualification statement must begin with a left parenthesis and end with a right parenthesis.
  - There must not be any blanks between the segment name or command codes, if used, and the left parenthesis.
  - The field name must be 8 bytes long. If the field name is less than 8 bytes, it must be left justified and padded on the right with blanks. The field name must have been defined for the specified segment type in the DBD the application program is using.
  - The relational operator follows the field name. It must be 2 bytes long and can be represented alphabetically or symbolically. The following table lists the relational operators.

Table 38. Relational operators

Symbolic	Alphabetic	Meaning
=b=	EQ	Equal to
>= or =>	GE	Greater than or equal to
<= or =<	LE	Less than or equal to
>b>	GT	Greater than
<b<	LT	Less than
¬= or =¬	NE	Not equal to

– The comparative value follows the relational operator. The length of this value must be equal to the length of the field that you specified in the field name. This length is defined in the DBD. The comparative value must include leading zeros for numeric values or trailing blanks for alphabetic values as necessary. The comparative value cannot include any parenthesis.

- If you are using multiple qualification statements within one SSA (Boolean qualification statements), the qualification statements must be separated by one of these symbols:

  *** or &** Dependent AND

  + or | Logical OR

  # Independent AND

  One of these symbols must appear between the qualification statements that the symbol connects.
- The last qualification statement must be followed by a right parenthesis.

An SSA created by the application program must not exceed the space allocated for the SSA in the PSB.

Related reading: For additional information about defining the PSB SSA size, see the explanation of the PSBGEN statement in *IMS Version 11 Database Utilities*.

SSA coding formats

Use the following formats to code segment search arguments in assembler language, C language, COBOL, Pascal, and PL/I.

Assembler language SSA definition examples

The following code example shows how you would define a qualified SSA without command codes. If you want to use command codes with this SSA, code the asterisk (*) and command codes between the 8-byte segment name field and the left parenthesis that begins the qualification statement.

```
*        CONSTANT AREA
:
SSANAME  DS    0CL26
ROOT     DC    CL8'ROOT    '
         DC    CL1'('
         DC    CL8'KEY     '
         DC    CL2' ='
NAME     DC    CLn'vv...v'
         DC    CL1')'
```

This SSA looks like this:

```
ROOT△△△△
(KEY△△△△△△
=vv...v)
```

C language SSA definition examples

An unqualified SSA that does not use command codes looks like this in C:

```
    const struct {
    char seg_name_u[8];
    char blank[1];
} unqual_ssa = {"NAME    ", " "};
```

You can use an SSA that is coded like this for each DL/I call that needs an unqualified SSA by supplying the name of the segment type you want during program execution. Note that the string size declarations are such that the C null terminators do not appear within the structure.

You can, of course, declare this as a single string:

```
const char unqual_ssa[] = "NAME     "; /* 8 chars + 1 blank  */
```

DL/I ignores the trailing null characters.

You can define SSAs in any of the ways explained for the I/O area.

The easiest way to create a qualified SSA is using the **sprintf** function. However, you can also define it using a method similar to that used by COBOL or PL/I.

The following is an example of a qualified SSA without command codes. To use command codes with this SSA, code the asterisk (*) and command codes between the 8-byte segment name field and the left parenthesis that begins the qualification statement.

```
struct {
    seg_name       char[8];
    seg_qual       char[1];
    seg_key_name   char[8];
    seg_opr        char[2];
    seg_key_value  char[n];
    seg_end_char   char[1];
} qual_ssa = {"ROOT    ", "(", "KEY     ", " =", "vv...vv", ")"};
```

Another way is to define the SSA as a string, using **sprintf**. Remember to use the preprocessor directive **#include <stdio.h>.**

```
char qual_ssa[8+1+8+2+6+1+1]; /* the final 1 is for the  */
                                  /* trailing '\0' of string */
sprintf(qual_ssa,
        "
        "ROOT", "KEY", "=", "vvvvv");
```

Alternatively, if only the value were changing, the **sprintf** call can be:

```
sprintf(qual_ssa,
        "ROOT    (KEY      =, "vvvvv");
      /* 12345678 12345678 */
```

In both cases, the SSA looks like this:

```
ROOT␣␣␣␣
(KEY␣␣␣␣␣␣
=vv...v)
```

COBOL SSA definition examples

An unqualified SSA that does not use command codes looks like this in COBOL:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
⋮
01  UNQUAL-SSA.
    02 SEG-NAME    PICTURE X(08)   VALUE '........'.
    02 FILLER      PICTURE X       VALUE ' '.
```

By supplying the name of the segment type you want during program execution, you can use an SSA coded like the one in this example for each DL/I call that needs an unqualified SSA.

Use a 01 level working storage entry to define each SSA that the program is to use. Then use the name you have given the SSA as the parameter in the DL/I call, in this case:

```
UNQUAL-SSA,
```

The following SSA is an example of a qualified SSA that does not use command codes. If you use command codes in this SSA, code the asterisk (*) and the command code between the 8-byte segment name field and the left parenthesis that begins the qualification statement.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
⋮
01  QUAL-SSA-MAST.
    02 SEG-NAME-M       PICTURE X(08)   VALUE 'ROOT    '.
    02 BEGIN-PAREN-M    PICTURE X       VALUE '('.
    02 KEY-NAME-M       PICTURE X(08)   VALUE 'KEY     '.
    02 REL-OPER-M       PICTURE X(02)   VALUE ' ='.
    02 KEY-VALUE-M      PICTURE X(n)    VALUE 'vv...v'.
    02 END-PAREN-M      PICTURE X       VALUE ')'.
```

The SSA looks like this:

```
ROOT△△△
(KEY△△△△△
=vv...v)
```

Java SSA definition examples

You can define SSAs by using either the IMS Universal DL/I driver or the classic Java APIs for IMS.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

The following information describes how to define SSAs using the classic Java APIs for IMS.

An unqualified SSA that uses command codes looks like this in Java:

```
// Construct an unqualified SSA for the Dealer segment
SSA dealerSSA = SSA.createInstance("Dealer");
```

You can use an SSA that is coded like the previous example for each DL/I call that needs an unqualified SSA by supplying the name of the segment type you want during program execution.

The following SSA is an example of a qualified SSA that does not use command codes. This SSA looks for all "Alpha" cars that were made in 1989. To use command codes with this SSA, use the method `addCommandCode` of the `SSA` class.

```
// Construct a qualified SSA for the Model segment
SSA modelSSA = SSA.createInstance("Model", "CarMake", SSA.EQUALS, "Alpha");
// Add an additional qualification statement
modelSSA.addQualification(SSA.AND, "CarYear", SSA.EQUALS, "1989");
```

Pascal SSA definition examples

An unqualified SSA that does not use command codes looks like this in Pascal:

```
type
  STRUCT = record
             SEG_NAME  : ALFA;
             BLANK     : CHAR;
           end;
const
  UNQUAL_SSA = STRUCT('NAME',' ');
```

You can also declare this SSA as a single string:

```
const
  UNQUAL_SSA = 'NAME     ';
```

The SSA shown in the following example is a qualified SSA that does not use command codes. If you use command codes in this SSA, code the asterisk (*) and the command code between the 8-byte segment name field and the left parenthesis that begins the qualification statement.

```
type
  STRUCT = record
             SEG_NAME       : ALFA;
             SEG_QUAL       : CHAR;
             SEG_KEY_NAME   : ALFA;
             SEG_OPR        : CHAR;
             SEG_KEY_VALUE  : packed array[1..n] of CHAR;
             SEG_END_CHAR   : CHAR;
           end;
const
  QUAL_SSA = STRUCT('ROOT','(','KEY',' =','vv...v',')');
```

This SSA looks like this:

```
ROOT△△△
(KEY△△△△△
=vv...v)
```

PL/I SSA definition examples

An unqualified SSA that does not use command codes looks like this in PL/I:

```
DCL 1  UNQUAL_SSA     STATIC UNALIGNED,
    2      SEG_NAME_U CHAR(8) INIT('NAME    '),
    2      BLANK      CHAR(1) INIT(' ');
```

You can use a SSA that is coded like this for each DL/I call that needs an unqualified SSA by supplying the name of the segment type you want during program execution.

In PL/I you define SSAs in structure declarations. The unaligned attribute is required for SSA data interchange with IMS. The SSA character string must reside contiguously in storage. For example, assignment of variable key values might cause IMS to construct an invalid SSA if the key value has changed the aligned attribute.

A separate SSA structure is required for each segment type that the program accesses because the value of the key fields differs among segment types. After you have initialized the fields (other than the key values), the SSA should not need to be changed again. You can define SSAs in any of the ways explained for the I/O area.

The following is an example of a qualified SSA without command codes. If you use command codes in this SSA, code the asterisk (*) and command codes between the 8-byte segment name field and the left parenthesis that begins the qualification statement.

```
DCL 1    QUAL_SSA             STATIC UNALIGNED,
         2       SEG_NAME                  CHAR(8) INIT('ROOT    '),
         2       SEG_QUAL                  CHAR(1) INIT('('),
         2       SEG_KEY_NAME              CHAR(8) INIT('KEY     '),
         2       SEG_OPR                   CHAR(2) INIT(' ='),
         2       SEG_KEY_VALUE             CHAR(n) INIT('vv...v'),
         2       SEG_END_CHAR              CHAR(1) INIT(')');
```

This SSA looks like this:

```
ROOT⌂⌂⌂
(KEY⌂⌂⌂⌂⌂
=vv...v)
```

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Data areas in GSAM databases

Generalized Sequential Access Method (GSAM) databases are available only to application programs that can run as batch programs, batch-oriented BMPs, transaction-oriented BMPS, or JBPs. The program communication block (PCB) mask and the record search argument (RSA) that you use in a GSAM database call have special formats.

GSAM DB PCB masks are slightly different from other DB PCB masks. The fields that are different are the length of the key feedback area and the key feedback area. Also, an additional field exists that gives the length of the record being retrieved or inserted when using undefined-length records.

The RSA is an 8-byte token for basic format data sets or 12-byte token for large format data sets that can be returned on `GN` and `ISRT` calls. The application program can save the RSA for use in a subsequent `GU` call.

Related concepts:

Chapter 19, "Processing GSAM databases," on page 305

Related reference:

"Specifying the DB PCB mask" on page 230

AIBTDLI interface

Use AIBTDLI as the interface between your application program and IMS.

Restriction: No fields in the AIB can be used by the application program except as defined by IMS.

When you use the AIBTDLI interface, you specify the program communication block (PCB) requested for the call by placing the PCB name (as defined by PSBGEN) in the resource name field of the AIB. You do not specify the PCB address. Because the AIB contains the PCB name, your application program can refer to the PCB name rather than the PCB address. Your application program does not need to know the relative PCB position in the PCB list. At completion of the call, the AIB returns the PCB address that corresponds to the PCB name passed by the application program.

The names of DB PCBs and alternate PCBs are defined by the user during PSBGEN. All I/O PCBs are generated with the PCB name ␣␣␣ For a generated program specification block (GPSB), the I/O PCB is generated with the PCB name IOPCB␣␣␣, and the modifiable alternate PCB is generated with the PCB name TPPCB1␣␣.

The ability to pass the PCB name means that you do not need to know the relative PCB number in the PCB list. In addition, the AIBTDLI interface enables your application program to make calls on PCBs that do not reside in the PCB list. The LIST= keyword, which is defined in the PCB macro during PSBGEN, controls whether the PCB is included in the PCB list.

The AIB resides in user-defined storage that is passed to IMS for DL/I calls that use the AIBTDLI interface. Upon call completion, IMS updates the AIB. Allocate at least 128 bytes of storage for the AIB.

Related concepts:

"PCB masks for GSAM databases" on page 305

Related reference:

"Application programming for PL/I" on page 376

"Application programming for Pascal" on page 373

"Application programming for C language" on page 368

"Application programming for assembler language" on page 365

"Assembler language application programming" on page 213

Language specific entry points

In your application program written in assembler language, C, COBOL, Pascal, or PL/I, control is passed from IMS through an *entry point*.

Your entry point must refer to the PCBs in the order in which they have been defined in the PSB. When you code each DL/I call, you must provide the PCB you want to use for that call. In all cases except CICS online, the list of PCBs that the program can access is passed to the program at its entry point. For CICS online, you must first schedule a PSB as described in the topic "System Service Call: PCB" in *IMS Version 11 Application Programming APIs*.

Application interfaces that use the AIB structure (AIBTDLI or CEETDLI), such as Java application interfaces, use the PCB name rather than the PCB structure and do not require the PCB list to be passed at entry to the application.

In a CICS online program, you do not obtain the address of the PCBs through an entry statement, but through the user interface block (UIB).

Leave the value blank if the application has been enabled for the IBM Language Environment® for z/OS & VM.

Assembler language entry point

You can use any name for the entry statement to an assembler language DL/I program. When IMS passes control to the application program, register 1 contains the address of a variable-length fullword parameter list. Each word in the list contains the address of a PCB. Save the content of register 1 before you overwrite it. IMS sets the high-order byte of the last fullword in the list to X'80' to indicate

the end of the list. Use standard z/OS linkage conventions with forward and backward chaining.

C language entry point

When IMS passes control to your program, it passes the addresses, in the form of pointers, for each of the PCBs that your program uses. The usual **argc** and **argv** arguments are not available to a program that is invoked by IMS. The IMS parameter list is made accessible by using the **__pcblist** macro. You can directly reference the PCBs by **__pcblist[0]**, **__pcblist[1]**, or you can define macros to give these more meaningful names. Note that I/O PCBs must be cast to get the proper type:

```
(IO_PCB_TYPE *)(__pcblist[0])
```

The entry statement for a C language program is the **main** statement.

```
#pragma runopts(env(IMS),plist(IMS))
#include <ims.h>

main()
{
.
.
.
}
```

The **env** option specifies the operating environment in which your C language program is to run. For example, if your C language program is invoked under IMS and uses IMS facilities, specify **env(IMS)**. The **plist** option specifies the format of the invocation parameters that is received by your C language program when it is invoked. When your program is invoked by a system support services program, the format of the parameters passed to your main program must be converted into the C language format: **argv**, **argc**, and **envp**. To do this conversion, you must specify the format of the parameter list that is received by your C language program. The **ims.h** include file contains declarations for `PCB` masks.

You can finish in three ways:

- End the main procedure without an explicit **return** statement.
- Execute a **return** statement from **main**.
- Execute an **exit** or an **abort** call from anywhere, or alternatively issue a **longjmp** back to **main**, and then do a normal return.

One C language program can pass control to another by using the **system** function. The normal rules for passing parameters apply; in this case, the **argc** and **argv** arguments can be used to pass information. The initial **__pcblist** is made available to the invoked program.

COBOL entry point

The procedure statement must refer to the I/O PCB first, then to any alternate PCB it uses, and finally to the DB PCBs it uses. The alternate PCBs and DB PCBs must be listed in the order in which they are defined in the PSB.

```
PROCEDURE DIVISION USING PCB-NAME-1 [,...,PCB-NAME-N]
```

In previous versions of IMS, `USING` might be coded on the entry statement to reference PCBs. However, IMS continues to accept such coding on the entry statement.

Recommendation: Use the procedure statement rather than the entry statement to reference the PCBs.

Pascal entry point

The entry point must be declared as a REENTRANT procedure. When IMS passes control to a Pascal procedure, the first address in the parameter list is reserved for Pascal's use, and the other addresses are the PCBs the program uses. The PCB types must be defined before this entry statement. The IMS interface routine PASTDLI must be declared with the GENERIC directive.

```
procedure ANYNAME(var SAVE: INTEGER;
                  var pcb1-name: pcb1-name-type[;
                  ...
                  var pcbn-name: pcbn-name-type]); REENTRANT;
procedure ANYNAME;
(* Any local declarations *)
  procedure PASTDLI; GENERIC;
begin
  (* Code for ANYNAME *)
end;
```

PL/I entry point

The entry statement must appear as the first executable statement in the program. When IMS passes control to your program, it passes the addresses of each of the PCBs your program uses in the form of pointers. When you code the entry statement, make sure you code the parameters of this statement as pointers to the PCBs, and not the PCB names.

```
anyname: PROCEDURE (pcb1_ptr [,..., pcbn_ptr]) OPTIONS (MAIN);
⋮
RETURN;
```

The entry statement can be any valid PL/I name.

CEETDLI, AIBTDLI, and AERTDLI interface considerations

The following considerations apply for CEETDLI, AIBTDLI, and AERTDLI.

The considerations for CEETDLI are:

- For PL/I programs, the CEETDLI entry point is defined in the CEEIBMAW include file. Alternatively, you can declare it yourself, but it must be declared as an assembler language entry (DCL CEETDLI OPTIONS(ASM);).
- For C language application programs, you must specify **env(IMS)** and **plist(IMS)**; these specifications enable the application program to accept the PCB list of arguments. The CEETDLI function is defined in <leawi.h>; the CTDLI function is defined in <ims.h>.

The considerations for AIBTDLI are:

- When using the AIBTDLI interface for C/MVS™, Enterprise COBOL, or PL/I language application programs, the language run-time options for suppressing abend interception (that is, NOSPIE and NOSTAE) must be specified. However, for Language Environment-conforming application programs, the NOSPIE and NOSTAE restriction is removed.
- The AIBTDLI entry point for PL/I programs must be declared as an assembler language entry (DCL AIBTDLI OPTIONS(ASM);).

- For C language applications, you must specify **env(IMS)** and **plist(IMS)**; these specifications enable the application program to accept the PCB list of arguments.

The considerations for AERTDLI are:

- When using the AERTDLI interface for C/MVS, COBOL, or PL/I language application programs, the language run-time options for suppressing abend interception (that is, NOSPIE and NOSTAE) must be specified. However, for Language Environment-conforming application programs, the NOSPIE and NOSTAE restriction is removed.
- The AERTDLI entry point for PL/I programs must be declared as an assembler language entry (DCL AERTDLI OPTIONS(ASM);).
- For C language applications, you must specify **env(IMS)** and **plis(IMS)**. These specifications enable the application program to accept the PCB list of arguments.
- AERTDLI must receive control with 31 bit addressability.

Related reference:

"Specifying the UIB (CICS online programs only)" on page 237

Program communication block (PCB) lists

In your application program, code your PCB or GPSB list in the following format.

PCB list format

The following example shows the general format of a PCB list.

```
[IOPCB]
[Alternate PCB ... Alternate PCB]
[DB PCB ... DB PCB]
[GSAM PCB ... GSAM PCB]
```

Each PSB must contain at least one PCB. An I/O PCB is required for most system service calls. An I/O PCB or alternate PCB is required for transaction management calls. (Alternate PCBs can exist in IMS TM.) DB PCBs for DL/I databases are used only with the IMS Database Manager under DBCTL. GSAM PCBs can be used with DCCTL.

Format of a GPSB PCB list

A generated program specification block (GPSB) takes this format:

```
[IOPCB]
[Alternate PCB]
```

A GPSB contains only an I/O PCB and one modifiable alternate PCB. (A modifiable alternate PCB enables you to change the destination of the alternate PCB while the program is running.) A GPSB can be used by all transaction management application programs, and permits access to the specified PCBs without the need for a specific PSB for the application program.

The PCBs in a GPSB have predefined PCB names. The name of the I/O PCB is IOPCB. The name of the alternate PCB is TPPCB1␢␢. The minimum size of the I/O work area that IMS generates for GPSBs in a DBCTL environment is 600 bytes.

PCB summary

If you intend to issue system service requests, be aware of the differences between I/O PCBs and alternate PCBs in various types of application programs.

DB Batch Programs
If `CMPAT=Y` is specified in PSBGEN, the I/O PCB is present in the PCB list; otherwise, the I/O PCB is not present, and the program cannot issue system service calls. Alternate PCBs are always included in the list of PCBs that IMS supplies to the program.

BMPs, MPPs, and IFPs
The I/O PCB and alternate PCBs are always passed to BMPs, MPPs, and IFPs.

The PCB list always contains the address of the I/O PCB, followed by the addresses of any alternate PCBs, followed by the addresses of the DB PCBs.

CICS Online Programs with DBCTL
If you specify the `IOPCB` option on the PCB call, the first PCB address in your PCB list is the I/O PCB, followed by any alternate PCBs, followed by the addresses of the DB PCBs.

If you do not specify the I/O PCB option, the first PCB address in your PCB list points to the first DB PCB.

The following table summarizes the I/O PCB and alternate PCB information.

Table 39. I/O PCB and alternate PCB information summary.

Environment	CALL DL/I	
	I/O PCB address in PCB list	**Alternate PCB address in PCB list**
MPP	Yes	Yes
IFP	Yes	Yes
BMP	Yes	Yes
DB Batch[1]	No	Yes
DB Batch[2]	Yes	Yes
TM Batch[3]	Yes	Yes
CICS DBCTL[4]	No	No
CICS DBCTL[5]	Yes	Yes

Notes:

1. CMPAT = N specified.
2. CMPAT = Y specified.
3. CMPAT = Option. Default is always to Y, even when CMPAT = N is specified.
4. SCHD request issued without the IOPCB or SYSSERVE option.
5. SCHD request issued with the IOPCB or SYSSERVE for a CICS DBCTL request or for a function-shipped request which is satisfied by a CICS system using DBCTL.

The AERTDLI interface

You can make database calls with AIBs in your ODBA applications using the AERTDLI interface.

Requirement: Allocate 264 bytes of storage for the AIB.

When you use the AERTDLI interface, the AIB used for database calls must be the same AIB as used for the APSB call. Specify the PCB that is requested for the call by placing the PCB name (as defined by PSBGEN) in the resource name field of the AIB. You do not specify the PCB address. Because the AIB contains the PCB name, your application can refer to the PCB name rather than to the PCB address. The AERTDLI call allows you to select PCBs directly by name rather than by a pointer to the PCB. At completion of the call, the AIB returns the PCB address that corresponds to the PCB name that is passed by the application program.

For PCBs to be used in a AERTDLI call, you must assign a name in PSBGEN, either with PCBNAME= or with the name as a label on the PCB statement. PCBs that have assigned names are also included in the positional pointer list, unless you specify LIST=NO. During PSBGEN, you define the names of the DB PCBs and alternate PCBs. All I/O PCBs are generated with the PCB name IOPCBbbb.

Because you pass the PCB name, you do not need to know the relative PCB number in the PCB list. In addition, the AERTDLI interface enables your application program to make calls on PCBs that do not reside in the PCB list. The LIST= keyword, which is defined in the PCB macro during PSBGEN, controls whether the PCB is included in the PCB list.

The AIB resides in user-defined storage that is passed to IMS for DL/I calls that use the AERTDLI interface. When the call is completed, the AIB is updated by IMS. Because some of the fields in the AIB are used internally by IMS, the same APSB AIB must be used for all subsequent calls for that PSB.

Language environments

IBM Language Environment provides the strategic execution environment for running your application programs written in one or more high level languages.

It provides not only language-specific run-time support, but also cross-language run-time services for your applications, such as support for initialization, termination, message handling, condition handling, storage management, and National Language Support. Many of Language Environment's services are accessible explicitly through a set of Language Environment interfaces that are common across programming languages; these services are accessible from any Language Environment-conforming program.

Language Environment-conforming programs can be compiled with the following compilers:

- IBM C++/MVS™
- IBM COBOL
- IBM PL/I

The CEETDLI interface to IMS

The language-independent CEETDLI interface to IMS is provided by Language Environment. It is the only IMS interface that supports the advanced error handling capabilities provided by Language Environment. The CEETDLI interface supports the same functionality as the other IMS application interfaces, and it has the following characteristics:

- The parmcount variable is optional.
- Length fields are 2 bytes long.
- Direct pointers are used.

Related reading: For more information about Language Environment, see *z/OS Language Environment Programming Guide*.

LANG= option on PSBGEN for PL/I compatibility

For IMS PL/I applications running in a compatibility mode that uses the PLICALLA entry point, you must specify LANG=PLI on the PSBGEN. Your other option is to change the entry point and add SYSTEM(IMS) to the EXEC PARM of the compile step so that you can specify LANG=blank or LANG=PLI on the PSBGEN. The following table summarizes when you can use LANG=blank and LANG=PLI.

Table 40. Using LANG= option in a Language Environment for PL/I compatibility

Compile exec statement is PARM=(...,SYSTEM(IMS)...	Entry point name is PLICALLA	Valid LANG= value
Yes	Yes	LANG=PLI
Yes	No	LANG=blank or LANG=PLI
No	No	**Note:** Not valid for IMS PL/I applications
No	Yes	LANG=PLI

PLICALLA is only valid for PL/I compatibility with Language Environment. If a PL/I application using PLICALLA entry at bind time is bound using Language Environment with the PLICALLA entry, the bind will work; however, you must specify LANG=PLI in the PSB. If the application is re-compiled using PL/I for z/OS & VM Version 1 Release 1 or later, and then bound using Language Environment Version 1 Release 2 or later, the bind will fail. You must remove the PLICALLA entry statement from the bind.

Special DL/I situations for IMS DB programming

Special cases during application programming for IMS DB include program scheduling against HALDBs, mixed language programming, using the extended addressing capabilities of z/OS, and setting COBOL compiler options for preloaded programs.

Application program scheduling against HALDBs

Application programs are scheduled against HALDBs the same way they are against non-HALDBs. Scheduling is based on the availability status of the HALDB master and is not affected by individual partition access and status.

The application programmer needs to be aware of changes to the handling of unavailable data for HALDBs. The feedback on data availability at PSB schedule time shows the availability of the HALDB master, not of the partitions. However, the error settings for data unavailability of a partition at the first reference to the partition during the processing of a DL/I call are the same as those of a non-HALDB, namely status code BA or pseudo ABENDU3303.

For example, if you issue the IMS /DBR command to half of the partitions to take them offline, the remaining partitions are available to the programs.

When an application program accesses a partition, that partition is considered to be in use by the application for the duration of that instance of the application. DBDUMP, DBRECOVERY, and START commands can operate against a partition currently not in use. The command is not processed for any partition that is being accessed by a BMP. A DFS0565I message is issued for partitions that are in use by a BMP. An exception to this rule is a partition where the accessing BMP issued a CHKP call and has not issued any subsequent DL/I calls. If an application attempts to access data from a stopped partition, a pseudo abend ABENDU3303 results or the application receives a BA status code. If the partition is started with the STA DB command before the application attempts to access data in that partition again, the DL/I call is processed successfully.

Mixed-language programming

When an application program uses the Language Environment language-independent interface, CEETDLI, IMS does not need to know the language of the calling program.

When the application program calls IMS in a language-dependent interface, IMS determines the language of the calling program according to the entry name that is specified in the CALL statement. That is, IMS assumes that the program is:

- Assembler language when the application program uses CALL ASMTDLI
- C language when the application program uses rc=CTDLI
- COBOL when the application program uses CALL CBLTDLI
- Pascal when the application program uses CALL PASTDLI
- PL/I when the application program uses CALL PLITDLI

For example, if a PL/I program calls an assembler language subroutine and the assembler language subroutine makes DL/I calls by using CALL ASMTDLI, the assembler language subroutine should use the assembler language calling convention, not the PL/I convention.

In this situation, where the I/O area uses the LLZZ format, LL is a halfword, not the fullword that is used for PL/I.

Extended addressing capabilities of z/OS

The two modes in z/OS with extended addressing capabilities are: the addressing mode (AMODE) and the residency mode (RMODE). IMS places no constraints on the RMODE and AMODE of an application program. The program can reside in the extended virtual storage area. The parameters that are referenced in the call can also be in the extended virtual storage area.

COBOL compiler options for preloaded programs

If you compile your COBOL program with the VS COBOL II compiler and preload it, you must use the COBOL compiler options RES and RENT.

Chapter 13. Establishing a DL/I interface from COBOL or PL/I

To establish a DL/I interface from COBOL or PL/I, use either the CBLTDLI procedure or the PLITDLI procedure.

CBLTDLI

The following control statements are necessary to establish a COBOL to DL/I interface. The block size of the following members must be less than or equal to 3200.

```
LIBRARY SDFSRESL(CBLTDLI)    DL/I LANGUAGE INTERFACE
LIBRARY SDFSRESL(DFHEI01)    HLPI LANGUAGE INTERFACE
LIBRARY SDFSRESL(DFHEI1)     HLPI LANGUAGE INTERFACE
```

PLITDLI

The following control statements are necessary to establish a PL/I to DL/I interface. The blocksize of the following members must be less than or equal to 3200.

```
LIBRARY SDFSRESL(PLITDLI)    DL/I LANGUAGE INTERFACE
LIBRARY SDFSRESL(DFHEI01)    HLPI LANGUAGE INTERFACE
LIBRARY SDFSRESL(DFHEI1)     HLPI LANGUAGE INTERFACE
ENTRY PLICALLA
```

PLITDLI is valid when using the PL/I Optimizing Compiler.

© Copyright IBM Corp. 1974, 2015

Chapter 14. Current position in the database after each call

Positioning means that DL/I tracks your place in the database after each call that you issue. By tracking your position in the database, DL/I enables you to process the database sequentially.

Current position after successful calls

Position is important when you process the database sequentially by issuing GN, GNP, GHN, and GHNP calls.

Current position is where IMS starts its search for the segments that you specify in the calls.

This section explains current position for successful calls. Current position is also affected by an unsuccessful retrieval or ISRT call.

Before you issue the first call to the database, the current position is the place immediately before the first root segment occurrence in the database. This means that if you issue an unqualified GN call, IMS retrieves the first root segment occurrence. It is the **next** segment occurrence in the hierarchy that is defined by the DB PCB that you referenced.

Certain calls cancel your position in the database. You can reestablish this position with the GU call. Because the CHKP and SYNC (commit point) calls cancel position, follow either of these calls with a GU call. The ROLS and ROLB calls also cancel your position in the database.

When you issue a GU call, your current position in the database does not affect the way that you code the GU call or the SSA you use. If you issue the same GU call at different points during program execution (when you have different positions established), you will receive the same results each time you issue the call. If you have coded the call correctly, IMS returns the segment occurrence you requested regardless of whether the segment is before or after the current position.

Exception: If a GU call does not have SSAs for each level in the call, it is possible for IMS to return a different segment at different points in your program. This is based on the position at each level.

For example, suppose you issue the following call against the data structure shown in the following figure.

```
GU   A⌂⌂⌂⌂⌂⌂⌂
AKEY⌂⌂⌂⌂
⌂
A1)
     B⌂⌂⌂⌂⌂⌂⌂
(BKEY⌂⌂⌂⌂
=⌂
B11)
     D⌂⌂⌂⌂⌂⌂⌂(DKEY⌂⌂⌂⌂D111)
```

The structure in the figure contains six segment types: A, B, C, D, E, and F. Figure 49 on page 258 shows one database record, the root of which is A1.

© Copyright IBM Corp. 1974, 2015

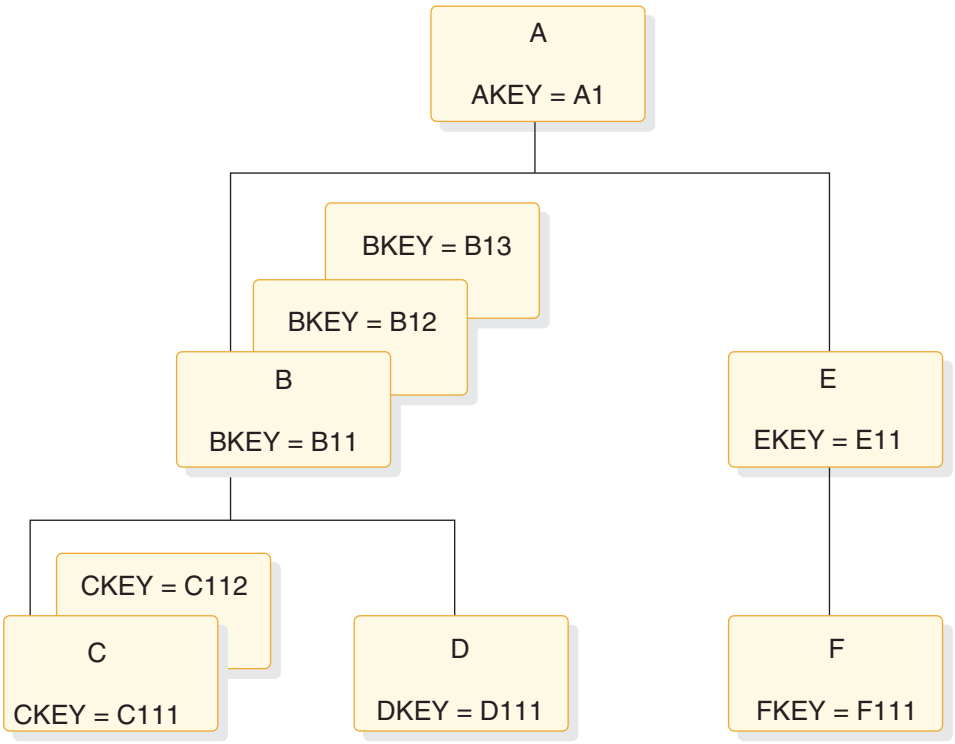


Figure 49. Current position hierarchy

When you issue this call, IMS returns the D segment with the key D111, regardless of where your position is when you issue the call. If this is the first call your program issues (and if this is the first database record in the database), current position before you issue the call is immediately before the first segment occurrence in the database—just before the A segment with the key of A1. Even if current position is past segment D111 when you issue the call (for example, just before segment F111), IMS still returns the segment D111 to your program. This is also true if the current position is in a different database record.

When you issue `GN` and `GNP` calls, current position in the database affects the way that you code the call and the SSA. That is because when IMS searches for a segment described in a `GN` or `GNP` call, it starts the search from current position and can only search forward in the database. IMS cannot look behind that segment occurrence to satisfy a `GN` or `GNP`. These calls can only move forward in the database when trying to satisfy your call, unless you use the F command code, the use of which is described in the topic "F Command Code" in *IMS Version 11 Application Programming APIs*.

If you issue a `GN` call for a segment occurrence that you have already passed, IMS starts searching at the current position and stops searching when it reaches the end of the database (resulting in a GB status code), or when it determines from your SSA that it cannot find the segment you have requested (GE status code).

Current position affects `ISRT` calls when you do not supply qualified SSAs for the parents of the segment occurrence that you are inserting. If you supply only the unqualified SSA for the segment occurrence, you must be sure that your position in the database is where you want the segment occurrence to be inserted.

Related concepts:

A command code (Application Programming APIs)

G command code (Application Programming APIs)
"Current position after unsuccessful calls" on page 263

Position after retrieval calls

After you issue any kind of successful retrieval call, position immediately follows the segment occurrence you just retrieved—or the lowest segment occurrence in the path if you retrieved several segment occurrences using the D command code. When you use the D command code in a retrieval call, a successful call is one that IMS completely satisfies.

For example, if you issue the following call against the database shown in the previous figure, IMS returns the C segment occurrence with the key of C111. Current position is immediately **after** C111. If you then issue an unqualified GN call, IMS returns the C112 segment to your program.

```
GU   Abbbbbbb
(AKEYbbbbEQA1)
     Bbbbbbbb(BKEYbbbbEQB11)
     Cbbbbbbb(CKEYbbbbEQC111)
```

Your current position is the same after retrieving segment C111, whether you retrieve it with GU, GN, GNP, or any of the Get Hold calls.

If you retrieve several segment occurrences by issuing a Get call with the D command code, current position is immediately after the lowest segment occurrence that you retrieved. If you issue the GU call as shown in the example above, but include the D command code in the SSA for segments A and B, the current position is still immediately after segment C111. C111 is the last segment that IMS retrieves for this call. With the D command code, the call looks like this:

```
GU   Abbbbbbb
(AKEYbbbbEQA1)
     Bbbbbbbb
(BKEYbbbbEQB11)
     Cbbbbbbb*D(CKEYbbbbEQC111)
```

You do not need the D command code on the SSA for the C segment because IMS always returns to your I/O area the segment occurrence that is described in the last SSA.

Position after DLET

After a successful DLET call, position immediately follows the segment occurrence you deleted. This is true when you delete a segment occurrence with or without dependents.

For example, if you issue the call shown in the following code example to delete segment C111, current position is immediately after segment C111. Then, if you issue an unqualified GN call, IMS returns segment C112.

```
GHU   A⌂⌂⌂⌂⌂⌂⌂
(AKEY⌂⌂⌂⌂
=⌂
A1)
      B⌂⌂⌂⌂⌂⌂⌂
(BKEY⌂⌂⌂⌂=⌂B11)
      C⌂⌂⌂⌂⌂⌂⌂(CKEY⌂⌂⌂⌂=⌂C111)
DLET
```

The following figure shows what the hierarchy looks like after this call. The successful DLET call has deleted segment C111.

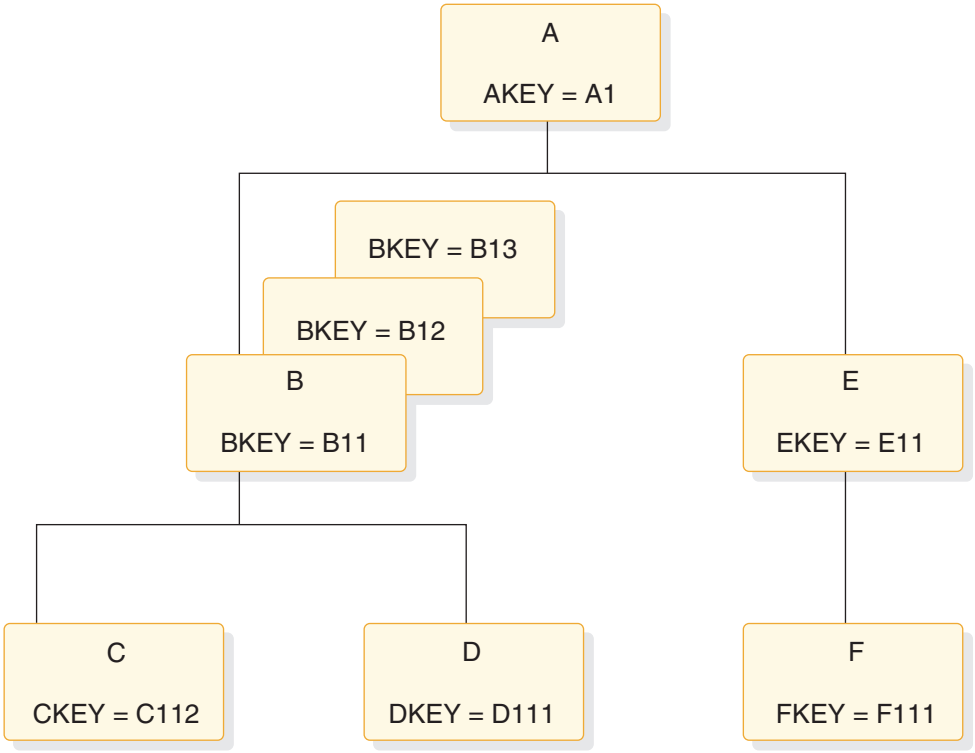


Figure 50. Hierarchy after deleting a segment

When you issue a successful DLET call for a segment occurrence that has dependents, IMS deletes the dependents, and the segment occurrence. Current position still immediately follows the segment occurrence you deleted. An unqualified GN call returns the segment occurrence that followed the segment you deleted.

For example, if you delete segment B11 in the hierarchy shown in the previous figure, IMS deletes its dependent segments, C112 and D111, as well. Current position immediately follows segment B11, just before segment B12. If you then issue an unqualified GN call, IMS returns segment B12. The following figure shows what the hierarchy looks like after you issued this call.

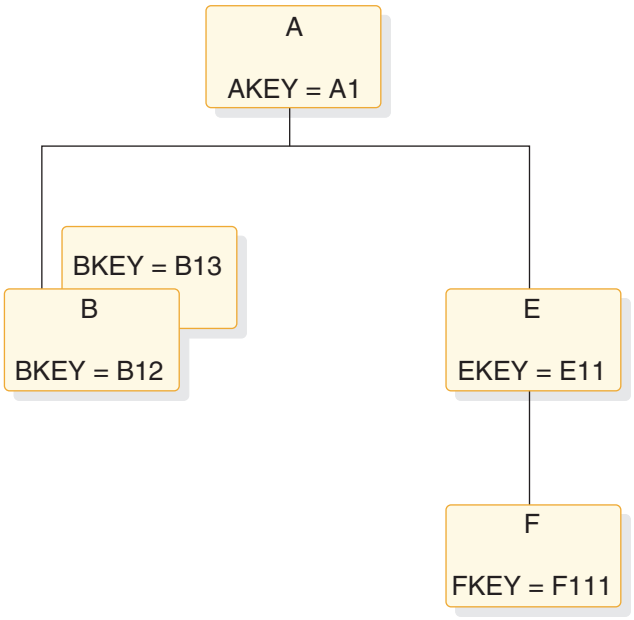


Figure 51. Hierarchy after deleting a segment and dependents

Because IMS deletes the segment's dependents, you can think of current position immediately following the last (lowest, right-most) dependent. In the example in the first figure, this immediately follows segment D111. But if you then issue an unqualified GN call, IMS still returns segment B12. You can think of position in either place—the results are the same either way. An exception to this can occur for a DLET that follows a GU path call, which returned a GE status code.

Related concepts:

"Current position after unsuccessful calls" on page 263

Position after REPL

A successful REPL call does not change your position in the database. Current position is just where it was before you issued the REPL call.

It immediately follows the lowest segment that is retrieved by the Get Hold call that you issued before the REPL call.

For example, if you retrieve segment B13 in the previous figure using a GHU instead of a GU call, change the segment in the I/O area, and then issue a REPL call, current position immediately follows segment B13.

Position after ISRT

After you add a new segment occurrence to the database, current position immediately follows the new segment occurrence.

For example, in the following figure, if you issue the following call to add segment C113 to the database, current position immediately follows segment C113. An unqualified GN call would retrieve segment D111.

```
ISRT   Abbbbbbb
(AKEYbbbb
=
```

If you are inserting a segment that has a unique key, IMS places the new segment in key sequence. If you are inserting a segment that has either a non-unique key or no key at all, IMS places the segment according to the rules parameter of the SEGM statement of the DBD for the database. the topic "ISRT Call" in *IMS Version 11 Application Programming APIs* explains these rules.

If you insert several segment occurrences using the D command code, current position immediately follows the lowest segment occurrence that is inserted.

For example, suppose you insert a new segment B (this would be B14), and a new C segment occurrence (C141), which is a dependent of B14. The following figure shows what the hierarchy looks like after these segment occurrences are inserted. The call to do this looks like this:

```
ISRT   Abbbbbbb
(AKEYbbbb
=
```

You do not need the D command code in the SSA for the C segment. On `ISRT` calls, you must include the D command code in the SSA for the only first segment you are inserting. After you issue this call, position immediately follows the C segment occurrence with the key of C141. Then, if you issue an unqualified `GN` call, IMS returns segment E11.

If your program receives an II status code as a result of an `ISRT` call (which means that the segment you tried to insert already exists in the database), current position is just **before** the duplicate of the segment that you tried to insert.

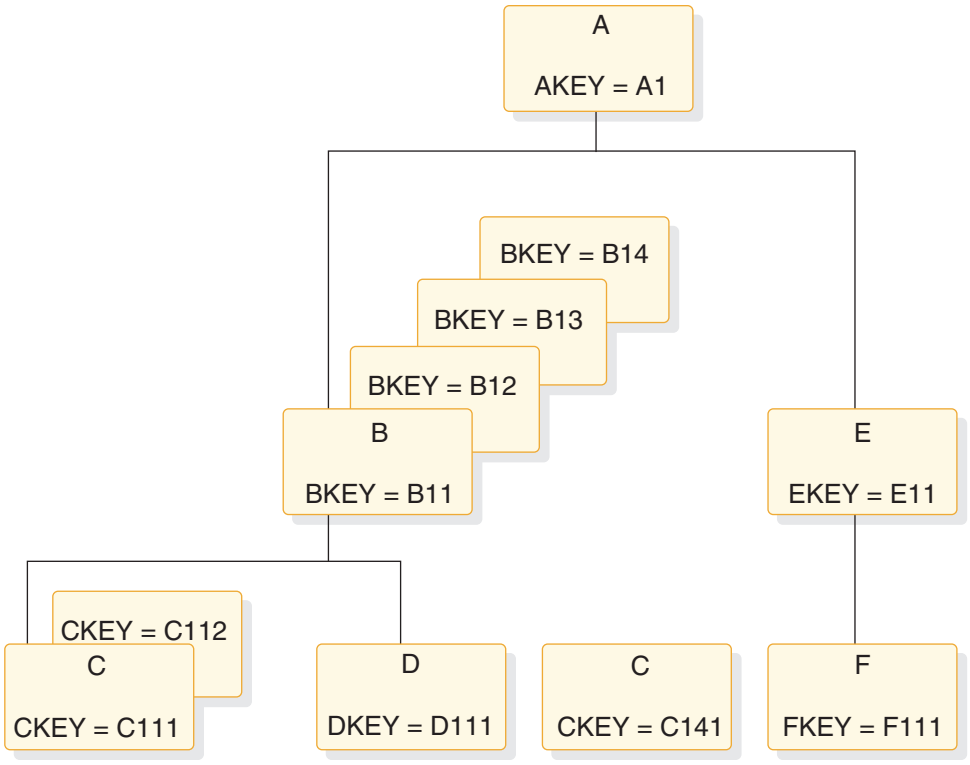


Figure 52. Hierarchy after adding new segments and dependents

Current position after unsuccessful calls

IMS establishes another kind of position when you issue retrieval and ISRT calls. This is position on one segment occurrence at each hierarchic level in the path to the segment that you are retrieving or inserting. Not every DL/I call that your program issues will be completely successful. When a call is unsuccessful, you should understand how to determine your position in the database after that call.

You need to know how IMS establishes this position to understand the U and V command codes described in the topic "General Command Codes for DL/I Calls" in *IMS Version 11 Application Programming APIs*. Also, you need to understand where your position in the database is when IMS returns a not-found status code to a retrieval or ISRT call.

Position after an unsuccessful DLET or REPL call

DLET and REPL calls do not affect current position. Your position in the database is the same as it was before you issued the call. However, an unsuccessful Get call or ISRT call does affect your current position.

To understand where your position is in the database when IMS cannot find the segment you have requested, you need to understand how DL/I determines that it cannot find your segment.

In addition to establishing current position after the lowest segment that is retrieved or inserted, IMS maintains a second type of position on one segment occurrence at each hierarchic level in the path to the segment you are retrieving or inserting.

For example, in the following figure, if you had just successfully issued the GU call with the SSA shown below, IMS has a position established at each hierarchic level.

```
GU   A⌂⌂⌂⌂⌂⌂⌂
(AKEY⌂⌂⌂⌂
=b⌂A1)
     B⌂⌂⌂⌂⌂⌂⌂
(BKEY⌂⌂⌂⌂⌂B11)
     C⌂⌂⌂⌂⌂⌂⌂(CKEY⌂⌂⌂⌂=⌂C111)
```

Now DL/I has three positions, one on each hierarchic level in the call:

- One on the A segment with the key A1
- One on the B segment with the key B11
- One on the C segment with the key C111

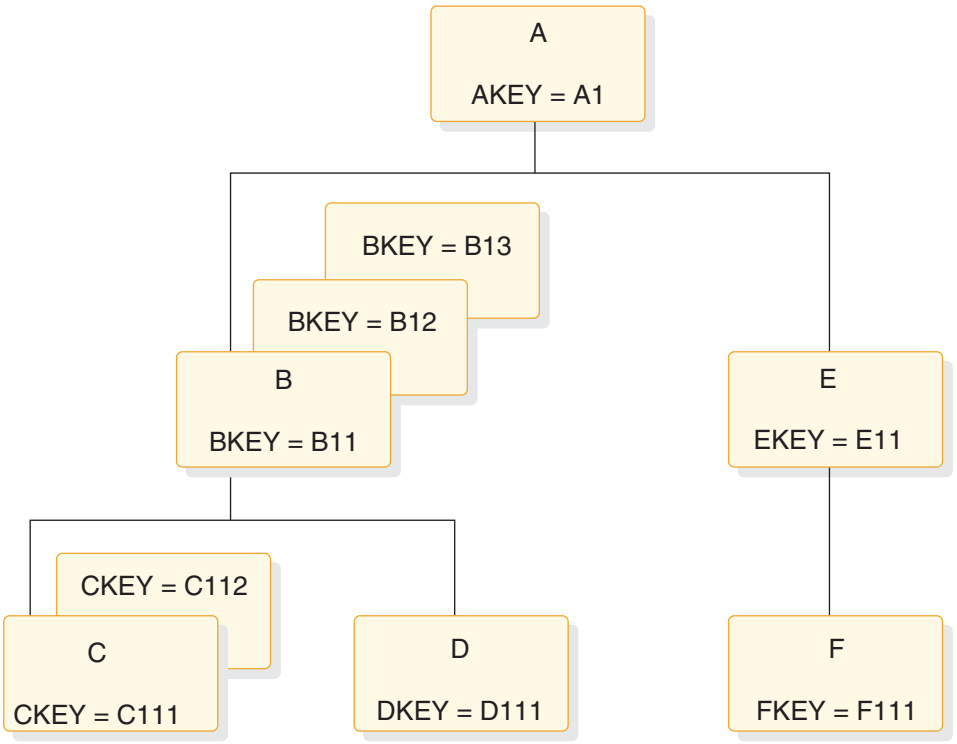


Figure 53. DL/I positions

When IMS searches for a segment occurrence, it accepts the first segment occurrence it encounters that satisfies the call. As it does so, IMS stores the key of that segment occurrence in the key feedback area.

Position after an unsuccessful retrieval or ISRT call

Current position after a retrieval or ISRT call that receives a GE status code depends on how far IMS got in trying to satisfy the SSA in the call. When IMS processes an ISRT call, it checks for each of the parents of the segment occurrence you are inserting. An ISRT call is similar to a retrieval call, because IMS processes the call level by level, trying to find segment occurrences to satisfy each level of the call. When IMS returns a GE status code on a retrieval call, it means that IMS was unable to find a segment occurrence to satisfy one of the levels in the call. When IMS returns a GE status code on an ISRT call, it means that IMS was unable to find one of the parents of the segment occurrence you are inserting. These are called not-found calls.

When IMS processes retrieval and ISRT calls, it tries to satisfy your call until it determines that it cannot. When IMS first tries to find a segment matching the description you have given in the SSA and none exists under the first parent, IMS tries to search for your segment under another parent. How you code the SSA in the call determines whether IMS can move forward and try again under another parent.

For example, suppose you issue the following GN call to retrieve the C segment with the key of C113 in the hierarchy shown in the previous figure.

```
GN   A␣␣␣␣␣␣␣(AKEY␣␣␣=␣A1)
     B␣␣␣␣␣␣␣(BKEY␣␣␣=␣B11)
     C␣␣␣␣␣␣␣(CKEY␣␣␣=␣C113)
```

When IMS processes this call, it searches for a C segment with the key equal to C113. IMS can only look at C segments whose parents meet the qualifications for the A and B segments. The B segment that is part of the path must have a key equal to B11, and the A segment that is part of the path must have a key equal to A1. IMS then looks at the first C segment. Its key is C111. The next C segment has a key of C112. IMS looks for a third C segment occurrence under the B11 segment occurrence. No more C segment occurrences exist under B11.

Because you have specified in the SSA that the A and B segment occurrences in C's path must be equal to certain values, IMS cannot look for a C segment occurrence with a key of C113 under any other A or B segment occurrence. No more C segment occurrences exist under the parent B11; the parent of C must be B11, and the parent of B11 must be A1. IMS determines that the segment you have specified does not exist and returns a not-found (GE) status code.

When you receive the GE status code on this call, you can determine where your position is from the key feedback area, which reflects the positions that IMS has at the levels it was able to satisfy—in this case, A1 and B11.

After this call, current position immediately follows the last segment occurrence that IMS examined in trying to satisfy your call—in this case, C112. Then, if you issue an unqualified `GN` call, IMS returns D111.

The current position after this call is different if A and B have non-unique keys. Suppose A's key is unique and B's is non-unique. After IMS searches for a C113 segment under B11 and is unable to find one, IMS moves forward from B11 to look for another B segment with a key of B11. When IMS does not find one, DL/I returns a GE status code. Current position is further in the database than it was when both keys were unique. Current position immediately follows segment B11. An unqualified `GN` call would return B12.

If A and B both have non-unique keys, current position after the previous call immediately follows segment A1. Assuming no more segment A1s exist, an unqualified `GN` call would return segment A2. If other A1s exist, IMS tries to find a segment C113 under the other A1s.

But suppose you issue the same call with a greater-than-or-equal-to relational operator in the SSA for segment B:

```
GU   A␣␣␣␣␣␣
(AKEY␣␣␣=>␣A1)
     B␣␣␣␣␣␣(BKEY␣␣␣=>B11)
     C␣␣␣␣␣␣(CKEY␣␣␣=>␣C113)
```

IMS establishes position on segment A1 and segment B11. Because A1 and B11 satisfy the first two SSAs in the call, IMS stores their keys in the key feedback area. IMS searches for a segment C113 under segment B11. None is found. But this time, IMS can continue searching, because the key of the B parent can be greater than or equal to B11. The next segment is B12. Because B12 satisfies the qualification for segment B, IMS places B12's key in the key feedback area. IMS then looks for a C113 under B12 and does not find one. The same thing happens for B13: IMS places the key of B13 in the key feedback area and looks for a C113 under B13.

When IMS finds no more B segments under A1, it again tries to move forward to look for B and C segments that satisfy the call under another A parent. But this time it cannot; the SSA for the A segment specifies that the A segment must be equal to A1. (If the keys were non-unique, IMS could look for another A1

segment.) IMS then knows that it cannot find a C113 under the parents you have specified and returns a GE status code to your program.

In this example, you have not limited the search for segment C113 to only one B segment, because you have used the greater-than-or-equal-to operator. The position is further than you might have expected, but you can tell what the position is from the key feedback area. The last key in the key feedback area is the key of segment B13. The current position of IMS immediately follows segment B13. If you then issue an unqualified GN call, IMS returns segment E11.

Each of the B segments that IMS examines for this call satisfies the SSA for the B segment, so IMS places the key of each in the key feedback area. But if one or more of the segments IMS examines does not satisfy the call, IMS does not place the key of that segment in the key feedback area. This means that the position in the database might be further than the position reflected by the key feedback area. For example, suppose you issue the same call, but you qualify segment B on a data field in addition to the key field. To do this, you use multiple qualification statements for segment B.

Assume the data field you are qualifying the call on is called BDATA. Assume the value you want is 14, but that only one of the segments, B11, contains a value in BDATA of 14:

```
GN   A⌂⌂⌂⌂⌂⌂⌂
(AKEY⌂⌂⌂⌂=⌂A1)
     B⌂⌂⌂⌂⌂⌂⌂(BKEY⌂⌂⌂⌂>=B11*BDATA⌂⌂⌂
=⌂
14)
     C⌂⌂⌂⌂⌂⌂⌂(CKEY⌂⌂⌂⌂=⌂C113)
```

After you issue this call, the key feedback area contains the key for segment B11. If you continue issuing this call until you receive a GE status code, the current position immediately follows segment B13, but the key feedback area still contains only the key for segment B11. Of the B segments IMS examines, only one of them (B11) satisfies the SSA in the call.

When you use a greater-than or greater-than-or-equal-to relational operator, you do not limit the search. If you get a GE status code on this kind of call, and if one or more of the segments IMS examines does not satisfy an SSA, the position in the database may be further than the position reflected in the key feedback area. If, when you issue the next GN or GNP call, you want IMS to start searching from the position reflected in the key feedback area instead of from its "real" position, you can either:

- Issue a fully qualified GU call to reestablish position to where you want it.
- Issue a GN or GNP call with the U command code. Including a U command code on an SSA tells IMS to use the first position it established at that level as qualification for the call. This is like supplying an equal-to relational operator for the segment occurrence that IMS has positioned on at that level.

For example, suppose that you first issue the GU call with the greater-than-or-equal-to relational operator in the SSA for segment B, and then you issue this GN call:

```
GN   A⌂⌂⌂⌂⌂⌂⌂*U
     B⌂⌂⌂⌂⌂⌂⌂*U
     C⌂⌂⌂⌂⌂⌂⌂⌂
```

The U command code tells IMS to use segment A1 as the A parent, and segment B11 as the B parent. IMS returns segment C111. But if you issue the same call

without the U command code, IMS starts searching from segment B13 and moves forward to the next database record until it encounters a B segment. IMS returns the first B segment it encounters.

Related concepts:

"Position after DLET" on page 259

Multiple processing

The order in which an application program accesses segments in a hierarchy depends on the purpose of the application program. Some programs access segments directly, others sequentially. Some application programs require that the program process segments in different hierarchic paths, or in different database records, in parallel.

If your program must process segments from different hierarchic paths or from different database records in parallel, using multiple positioning or multiple PCBs can simplify the program's processing. For example:

- Suppose your program must retrieve segments from **different hierarchic paths** alternately: for example, in the following figure, it might retrieve B11, then C11, then B12, then C12, and so on. If your program uses **multiple positioning**, IMS maintains positions in both hierarchic paths. Then the program is not required to issue `GU` calls to reset position each time it needs to retrieve a segment from a different path.
- Suppose your program must retrieve segments from **different database records** alternately: for example, it might retrieve a B segment under A1, and then a B segment under another A root segment. If your program uses **multiple PCBs**, IMS maintains positions in both database records. Then the program does not have to issue `GU` calls to reset position each time it needs to access a different database record.

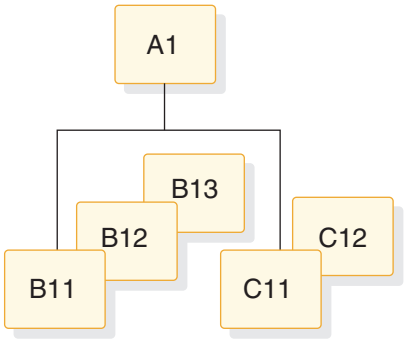


Figure 54. Multiple processing

Multiple positioning

When you define the PSB for your application program, you have a choice about the kind of positioning you want to use: single or multiple. All of the examples used so far, and the explanations about current position, have used single positioning.

Specify the kind of position you want to use for each PCB on the `PCB` statement when you define the PSB. The `POS` operand for a DEDB is disregarded. DEDBs support multiple positioning only.

Single positioning
IMS maintains position in **one** hierarchic path for the hierarchy that is defined by that PCB. When you retrieve a segment, IMS clears position for all dependents and all segments on the same level.

Multiple positioning
IMS maintains position in **each** hierarchic path in the database record that is being accessed. When you retrieve a segment, IMS clears position for all dependents but keeps position for segments at the same level. You can process different segment types under the same parent in parallel.

For example, suppose you issue these two calls using the hierarchy shown in the following figure:

```
GU    Abbbbbbb
(AKEYbbbb
=bA1)
      Bbbbbbbb
(BKEYYbbbb
=b
B11)
      Cbbbbbbb
(CKEYYbbbb
=b
C111)

GN    Ebbbbbbb(EKEYYbbbb
=b
E11)
```

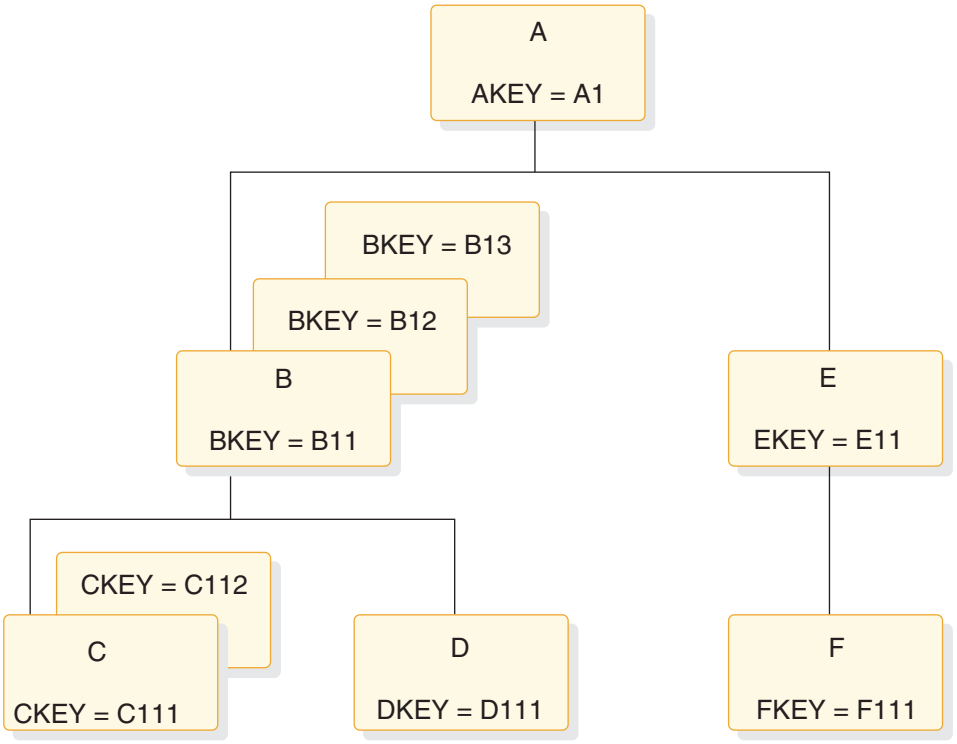


Figure 55. Multiple positioning hierarchy

After issuing the first call with single positioning, IMS has three positions established: one on A1, one on B11, and one on C111. After issuing the second call, the positions on B11 and C111 are canceled. Then IMS establishes positions on A1 and E11.

After issuing the first call with single and multiple positioning, IMS has three positions established: one on A1, one on B11, and one on C111. However, after issuing the second call, single positioning cancels positions on B11 and C111 while multiple positioning retains positions on B11 and C111. IMS then establishes positions on segments A1 and E11 for both single and multiple positioning.

After issuing the first call with multiple positioning, IMS has three positions established (just as with single positioning): one on A1, one on B11, and one on C111. But after issuing the second call, the positions on B11 and C111 are retained. In addition to these positions, IMS establishes position on segments A1 and E11.

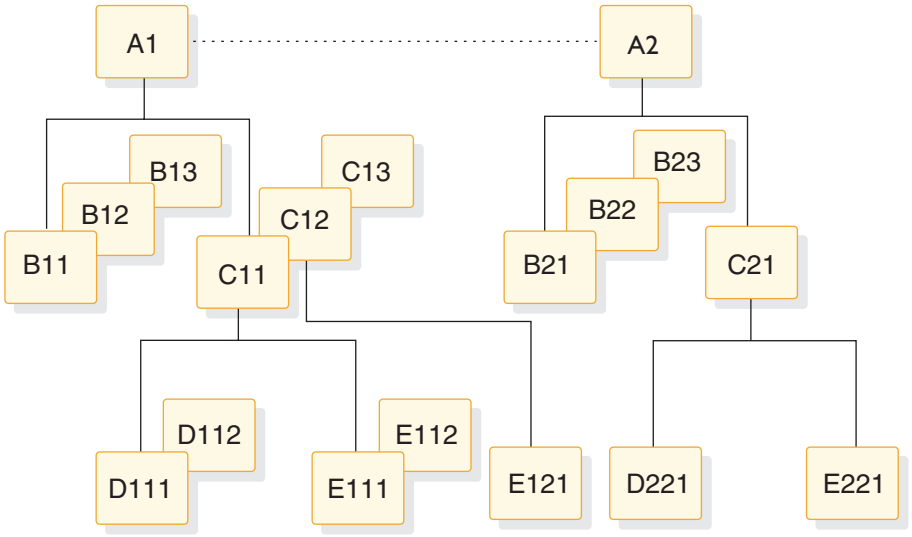


Figure 56. Single and multiple positioning hierarchy

The examples that follow compare the results of single and multiple positioning using the hierarchy in the following figure.

Table 41. Results of single and multiple positioning with DL/I calls

Sequence	Result of Single Positioning	Result of Multiple Positioning
Example 1		
GU (where AKEY equals A1)	A1	A1
GNP B	B11	B11
GNP C	C11	C11
GNP B	Not found	B12
GNP C	C12	C12
GNP B	Not found	B13
GNP C	C13	C13
GNP B	Not found	Not found
GNP C	Not found	Not found
Example 2		
GU A (where AKEY equals A1)	A1	A1
GN B	B11	B11
GN C	C11	C11

Table 41. Results of single and multiple positioning with DL/I calls (continued)

Sequence	Result of Single Positioning	Result of Multiple Positioning
GN B	B21	B12
GN C	C21	C12
Example 3		
GU A (where AKEY equals A1)	A1	A1
GN C	C11	C11
GN B	B21	B11
GN B	B22	B12
GN C	C21	C12
Example 4		
GU A (where AKEY equals A1)	A1	A1
GN B	B11	B11
GN C	C11	C11
GN D	D111	D111
GN E	E111	E111
GN B	B21	B12
GN D	D221	D112
GN C	C under next A	C12
GN E	E under next A	E121

Multiple positioning is useful when you want to examine or compare segments in two hierarchic paths. It lets you process different segment types under the same parent in parallel. Without multiple positioning, you would have to issue GU calls to reestablish position in each path.

Advantages of using multiple positioning

The advantages of using multiple positioning include the following:

- You might be able to design your program with greater data independence than you would using single positioning. You can write application programs that use GN and GNP calls, and GU and ISRT calls with missing levels in their SSAs, independent of the relative order of the segment types being processed. If you improve your program's performance by changing the relative order of segment types and all of the application programs that access those segment types use multiple positioning, you could make the change without affecting existing application programs. To do this without multiple positioning, the program would have to use GN and GNP calls, and GU and ISRT calls with incompletely specified SSAs.
- Your program can process dependent segment types in parallel (it can switch back and forth between hierarchic paths without reissuing GU calls to reset position) more efficiently than is possible with single positioning. You indicate to IMS the hierarchic path that contains the segments you want in your SSAs in the call. IMS uses the position established in that hierarchic path to satisfy your call. The control blocks that IMS builds for each kind of positioning are the same. Multiple positioning does not require more storage, nor does it have a big impact on performance.

Keep in mind that multiple positioning might use more processor time than single positioning, and that multiple positioning cannot be used with HSAM databases.

How multiple positioning affects your program

Multiple positioning affects the order and structure of your DL/I calls.

GU and ISRT

The only time multiple positioning affects GU and ISRT calls is when you issue these calls with missing SSAs in the hierarchic path. When you issue a GU or ISRT call that does not contain an SSA for each level in the hierarchic path, IMS builds the SSA for the missing levels according to the current position:

- If IMS has a position established at the missing level, the qualification IMS uses is derived from that position, as reflected in the DB PCB.
- If no position is established at the missing level, IMS assumes a segment type for that level.
- If IMS moves forward from a position that is established at a higher level, it assumes a segment type for that level.

Because IMS builds the missing qualification based on current position, multiple positioning makes it possible for IMS to complete the qualification independent of current positions that are established for other segment types under the same parent occurrence.

DLET and REPL with multiple positioning

Multiple positioning does not affect DLET or REPL calls; it only affects the Get Hold calls that precede them.

Qualified GN and GNP calls

When your program issues a GN or GNP call, IMS tries to satisfy the call by moving forward from current position. When you use multiple positioning, more than one current position exist: IMS maintains a position at each level in **all** hierarchic paths, instead of at each level in **one** hierarchic path. To satisfy GN and GNP calls with multiple positioning, IMS moves forward from the current position in the path that is referred to in the SSA.

Mixing qualified and unqualified GN and GNP calls

Although multiple positioning is intended to be used with qualified calls for parallel processing and data independence, you may occasionally want to use unqualified calls with multiple positioning. For example, you may want to sequentially retrieve all of the segment occurrences in a hierarchy, regardless of segment type.

Recommendation: Limit unqualified calls to GNP calls in order to avoid inconsistent results. Mixing qualified and unqualified SSAs may be valid for parallel processing, but doing so might also decrease the program's data independence.

There are three rules that apply to mixing qualified and unqualified GN and GNP calls:

1. When you issue an unqualified GN or GNP, IMS uses the position that is established by the preceding call to satisfy the GN or GNP call. For example:

Your program issues these calls:	DL/I returns these segments:
GU A (where AKEY = A1)	A1
GN B	B11
GN E	E11
GN	F111

When your program issues the unqualified GN call, IMS uses the position that is established by the last call, the call for the E segment, to satisfy the unqualified call.

2. After you successfully retrieve a segment with an unqualified GN or GNP, IMS establishes position in only one hierarchic path: the path containing the segment just retrieved. IMS cancels positions in other hierarchic paths. IMS establishes current position on the segment that is retrieved and sets parentage on the parent of the segment that is retrieved. If you issue a qualified call for a segment in a different hierarchic path after issuing an unqualified call, the results are unpredictable. For example:

Your program issues these calls:	DL/I returns these segments:
GU A (where AKEY = A1)	A1
GN B	B11
GN E	E11
GN	F111
GN B	unpredictable

When you issue the unqualified GN call, IMS no longer maintains a position in the other hierarchic path, so the results of the GN call for the B segment are unpredictable.

3. If you issue an unqualified GN or GNP call and IMS has a position established on a segment that the unqualified call might encounter, the results of the call are unpredictable. Also, when you issue an unqualified call and you have established position on the segment that the call "should" retrieve, the results are unpredictable.

   For example:

Your program issues these calls:	DL/I returns these segments:
GU A (where AKEY = A1)	A1
GN E	E11
GN D	D111
GN B	B12
GN B	B13
GN	E11 (The only position IMS has is the one established by the GN call.)

In this example, IMS has a position established on E11. An unqualified GN call moves forward from the position that is established by the previous call. Multiple positions are lost; the only position IMS has is the position that is established by the GN call.

To summarize these rules:

1. To satisfy an unqualified GN or GNP call, IMS uses the position established in the last call for that PCB.
2. If an unqualified GN or GNP call is successful, IMS cancels positions in all other hierarchic paths. Position is maintained only within the path of the segment retrieved.

Resetting position with multiple positioning

To reset position, your program issues a GU call for a root segment. If you want to reset position in the database record you are currently processing, you can issue a GU call for that root segment, but the GU call cannot be a path call.

Example: Suppose you have positions established on segments B11 and E11. Your program can issue one of the calls below to reset position on the next database record.

Issuing this call causes IMS to cancel all positions in database record A1:

```
GU   Abbbbbbb
AKEYbbbb
=b
A2)
```

Or, if you wanted to continue processing segments in record A1, you issue this call to cancel all positions in record A1:

```
GU    Abbbbbbb
AKEYbbbb
=b
A1)
```

Issuing this call as a path call does not cancel position.

Multiple DB PCBs

When a program has multiple PCBs, it usually means that you are defining views of several databases, but this also can mean that you need several positions in one database record. Defining multiple PCBs for the same hierarchic view of a database is another way to maintain more than one position in a database record.

Using multiple PCBs also extends what multiple positioning does, because with multiple PCBs you can maintain positions in two or more database records and within two or more hierarchic paths in the same record.

For example, suppose you were processing the database record for Patient A. Then you wanted to look at the record for Patient B and also be able to come back to your position for Patient A. If your program uses multiple PCBs for the medical hierarchy, you issue the first call for Patient A using PCB1 and then issue the next call, for Patient B, using PCB2. To return to Patient A's record, you issue the next call using PCB1, and you are back where you left off in that database record.

Using multiple PCBs can decrease the number of Get calls required to maintain position and can sometimes improve performance. Multiple PCBs are particularly useful when you want to compare information from segments in two or more database records. However, the internal control block requirements increase with each PCB that you define.

You can use the AIBTDLI interface with multiple PCBs by assigning different PCBNAMEs to the PCBs during PSB generation. Just as multiple PCBs must have

different addresses in the PSB PCBLIST, multiple PCBs must have different PCBNAMEs when using the AIBTDLI interface. For example, if your application program issues DL/I calls against two different PCBs in a list that identifies the same database, you achieve the same effect with the AIBTDLI interface by using different PCBNAMEs on the two PCBs at PSB generation time.

Chapter 15. Using IMS application program sync points

IMS application programs can (and should) take checkpoints. These checkpoints and system sync points can affect IMS operations.

Commit process

During the synchronization point (sync point) processing for an application, IMS creates a log record to establish commitment of database changes and availability of output messages. The commit process is not complete until IMS physically writes this log record to the OLDS because an incomplete set of database change and message records exist on the log for system restart.

The commit processes work differently for DL/I and Fast Path applications. For DL/I, IMS makes database changes in the buffer pool at the time of a DL/I call, and can write the changes to disk before the commit point. If you restart the system, IMS backs out these uncommitted changes by using the log. IMS stores inserted message segments in the message queue and must similarly discard them.

For Fast Path, IMS keeps all changes in storage until it physically logs the commit record. Only then does IMS write database changes to DASD and send output messages. Because no changes appear on external storage (except for the log) until the commit record is written, IMS does not perform backout processing for the database. IMS discards the updates in storage. With Fast Path, system restart ensures that IMS writes committed updates to DASD and sends output messages.

Relationship between checkpoints and sync points

IMS tracks all checkpoints and sync points. IMS usually uses a sync point during recovery, but returns to the checkpoint in the following situations: In the following figure, for example, if a system-wide failure occurs in the DB/DC environment just after the MTO takes a system checkpoint but just before program B commits (assuming that program A has not made any updates since its last commit), IMS must return to the system checkpoint before Beta started.

- For a full recovery in the DB/DC environment, IMS returns to the earliest of either the checkpoint before the current checkpoint or the checkpoint before the first uncommitted application program update.
- For a full recovery in the DBCTL environment, IMS always returns to the checkpoint before the first uncommitted application program update.
- For a full recovery in the DCCTL environment, IMS always returns to the checkpoint before the latest system checkpoint.
- In the DB/DC or DCCTL environments, if a BUILDQ is requested on the restart, IMS returns to the last SNAPQ or DUMPQ checkpoint. IMS returns to this checkpoint even if it is older than the checkpoint normally needed for the restart.

© Copyright IBM Corp. 1974, 2015

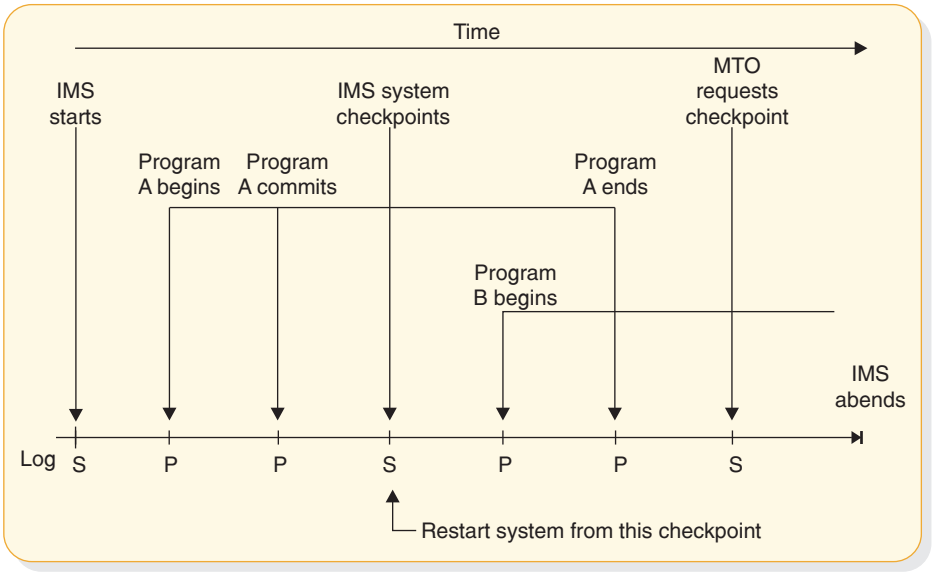


Figure 57. Independence of system checkpoints and application sync points

Synchronization point processing in CPI Communications-driven programs

For CPI Communications-driven programs running under Advanced Program-to-Program Communications for IMS (APPC/IMS), the application programs control their own sync point processing. An application program can issue certain CPI Resource Recovery calls: `SRRCMIT` calls to commit data and `SRRBACK` calls to back out data. The protected resources managed by IMS (local) include:

- IMS TM message-queue messages
- IMS DB databases
- DB2 for z/OS databases

The highest level of synchronization supported for a conversation is SYNCPT, so CPI Communications-driven applications can have protected conversations.

Sync point and resource manager

IMS can be either the sync point manager or the resource manager, depending on the setting of the sync point level. For SYNC_LEVEL=NONE or CONFIRM, IMS is the sync point manager, but for SYNC_LEVEL=SYNCPT, z/OS Resource Recovery Services (RRS) is the sync point manager and IMS is the resource manager.

Two-phase commit in the synchronization process

Application programs in a DBCTL, DCCTL, DB/DC, APPC/IMS, or OTMA environment can be involved in a two-phase commit process to record a sync point. At the completion of a two-phase commit, the resource manager commits database and message changes.

The two phases are:

1. Phase 1, in which the sync-point coordinator directs sync point preparation and asks the connected resource managers whether updates to connected databases can be committed.

The sync-point coordinator can be:

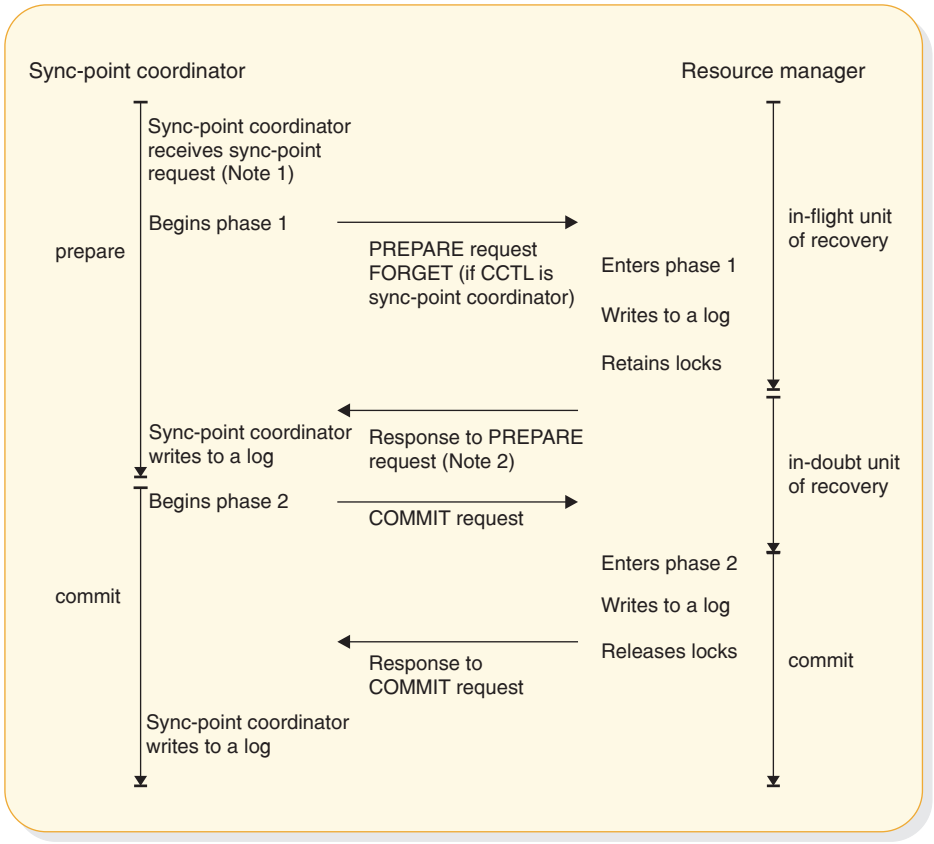
- An IMS DB/DC subsystem for its resource managers and attached databases.
- An IMS DCCTL subsystem for attached databases.
- A Coordinator Controller (CCTL) subsystem for units of work associated with the CCTL region. IMS DB acts as a resource manager when connected to a CCTL and also when accessed by ODBA application programs through the Open Database Access (ODBA) interface.
- z/OS Resource Recovery Services (RRS) for its protected conversations with APPC/IMS applications programs or OTMA clients. IMS acts as a resource manager when connected to RRS.

2. Phase 2, in which the sync-point coordinator directs commit or abort processing and states that the resources must either be committed or aborted.

   In the DBCTL environment, if an application program makes no update DL/I calls or makes only inquiry-type DL/I calls, the CCTL requests a "forget" response to Phase 1 (if forget processing has been enabled). This means that only a limited Phase 2 occurs for that application program because no database resources have been altered. See *IMS Version 11 Exit Routines* for details on how to enable forget processing.

   The sync-point coordinator can request an abort without a Phase 1.

The following figure shows the two phases of the sync-point cycle for an IMS DBCTL environment and describes the activities taking place.



Notes:

1. If the resource manager indicates that it cannot commit the updates, the sync-point coordinator should abort the unit of recovery, and the rest of this figure does not apply.
2. If the sync-point coordinator tells the resource manager to commit the updates, then it **must** commit.

Figure 58. Two-phase commit process

Unit of recovery

A *unit of recovery* (UOR) is the work done by a thread (connection between a resource-manager control region and a sync-point coordinator) during a sync-point interval, that is between two sync points.

In-flight unit of recovery

The unit of recovery is said to be *in-flight* from its creation or its last sync point until the resource manager logs the end of Phase 1. If a resource manager fails before or during Phase 1 and is subsequently restarted, IMS aborts all database updates.

In-doubt unit of recovery for DBCTL connected to CCTL

From the time that the resource manager issues its response to the PREPARE request (the completion of Phase 1), to the time it receives a COMMIT or ABORT request from the CCTL, units of recovery are said to be *in-doubt*. When the

resource manager is restarted after a failure, it tells the CCTL which in-doubt UORs exist, if any. The CCTL then takes action to resolve these in-doubt UORs. This is called resolve in-doubt processing, or resynchronization. If a CCTL cannot resolve all in-doubt UORs, you can use IMS or CCTL commands to display the units of recovery and take appropriate actions for committing or aborting them.

Recovery tokens for DBCTL connected to CCTL

A *recovery token* is a 16-byte identifier for each unit of recovery. The resource manager validates the recovery token to protect against duplication of units of recovery. In the DBCTL environment, you can display the recovery token using the IMS `/DISPLAY CCTL` command. The recovery token is the primary identifier used by DBRC, which performs unit-of-recovery management. DBRC keeps track of backouts that are appropriate for the Batch Backout utility to perform.

Recoverable in-doubt structure

An IMS DBCTL subsystem builds a recoverable in-doubt structure (RIS) for each in-doubt UOR when any of the following occurs:

- A CCTL fails
- A CCTL thread fails
- A resource manager fails

The resource manager uses a recoverable in-doubt structure during reconnecting to the CCTL if in-doubt UORs existed when either the CCTL or the resource manager failed. IMS logs all recoverable in-doubt structures during system checkpoints.

A recoverable in-doubt structure contains the following information:

- The recovery token in a residual recovery element (RRE)
- Changed data records in an in-doubt extended error queue element (IEEQE)
- An indication of data that is inaccessible because of unresolved in-doubt UORs
- Links to other recoverable in-doubt structures using extended error queue element (EEQE) queue elements (EQELs)

DBCTL single-phase commit

A CCTL communicating with just one resource manager (IMS DBCTL subsystem) can request a sync point using just a single phase. If the CCTL communicates with more than one resource manager, it must use the two-phase commit process.

When the CCTL decides to commit a UOR, it can request a single-phase sync point. Single-phase commit can affect the recoverability of in-doubt data. A transaction is only in-doubt for the short time between the sync-point request and DBCTL's commit. IMS can recover in-doubt data after a thread failure during single-phase commit, but cannot recover in-doubt data after a subsystem failure.

Sync-point log records

During the two-phase commit process, IMS creates log records to establish the commitment of database changes. All these log records can be used by the IMS Change Accumulation and recovery utilities.

All online log records involving the sync-point cycle contain a recovery token. This token ensures that IMS can recover and restart each unit of recovery. The sequence of log records for a unit of recovery reveals the sync-point cycle that it followed.

IMS logs the following records during the sync-point process:

Log record
Description

X'08' Schedule record

X'07' Unschedule (terminate) record

X'0A08'
CPI Communications-driven application program schedule record

X'0A07'
CPI Communications-driven application program unschedule (terminate) record

X'5945'
Fast Path 64-bit buffer usage

X'5937'
Fast Path start commit

X'5938'
Fast Path start abort

X'5610'
Start of Phase 1

X'5611'
End of Phase 1

X'3730'
Start of Phase 2 Commit

X'5612'
End of Phase 2 Commit

X'3801'
Start of abort

X'4C01'
End of abort

X'5607'
Start unit of recovery

X'5613'
Recoverable in-doubt structure created

X'5614'
Recoverable in-doubt structure deleted

Sync points with a data-propagation manager

When using a data-propagation manager (such as the IMS DataPropagator) to update DB2 for z/OS databases synchronously with IMS DL/I databases, the updates to the DB2 for z/OS databases are committed (or aborted) at the same time as the IMS updates. This provides consistency between the database management subsystems. IMS DB/DC, DCCTL, and DBCTL (BMP regions only) support the IMS Data Capture exit routine.

Restriction: In an IMS DBCTL environment, the data-propagation manager is available only for BMP regions.

For more information about the IMS DataPropagator, go to the following web URL: http://www.ibm.com/software/data/db2imstools/imstools/imsdprop.html

Chapter 16. Recovering databases and maintaining database integrity

You can issue checkpoints, restart programs, and maintain database integrity in your application programs.

Java applications running in Java batch processing (JBP) regions can issue symbolic checkpoint and restart calls by using the IMS Java dependent region resource adapter.

Related concepts:

"Developing JBP applications with the IMS Java dependent region resource adapter" on page 652

Issuing checkpoints

Two kinds of checkpoint (CHKP) calls exist: the basic CHKP and the symbolic CHKP. All IMS programs and CICS shared database programs can issue the basic CHKP call; only BMPs and batch programs can use either call.

IMS Version 11 Application Programming APIs explains when and why you should issue checkpoints in your program. Both checkpoint calls cause a loss of database position when the call is issued, so you must reestablish position with a GU call or some other method. You cannot reestablish position in the middle of non-unique keys or nonkeyed segments.

Restriction: You must not specify CHKPT=EOV on any DD statement to take an IMS checkpoint.

Some differences exist if you issue the same call sequence against a full-function database or a DEDB, and an MSDB.

Depending on the database organization, a CHKP call can result in the database position for the PCB being reset. When the CHKP call is issued, the locks held by the program are released. Therefore, if locks are necessary for maintaining your database position, the position is reset by the CHKP call. Position is reset in all cases except those in which the organization is either GSAM (locks are not used) or DEDB, and the CHKP call is issued after a GC status code. For a DEDB, the position is maintained at the unit-of-work boundary.

Issuing a CHKP resets the destination of the modifiable alternate PCB.

Related Reading: For more information on CHKP calls, see the topic "CHKP (Basic) Call" and the topic "CHKP (Symbolic) Call" in *IMS Version 11 Application Programming APIs* .

Related concepts:

"Commit-point processing in MSDBs and DEDBs" on page 327

© Copyright IBM Corp. 1974, 2015

Restarting your program from the latest checkpoint

If you use basic checkpoints instead of symbolic checkpoints, provide the necessary code to restart the program from the latest checkpoint if the program terminates abnormally.

One way to restart the program from the latest checkpoint is to store repositioning information in a HDAM or PHDAM database. With this method, your program writes a database record containing repositioning information to the database each time a checkpoint is issued. Before your program terminates, it should delete the database record.

For more information on the XRST call, see the topic "XRST Call" in *IMS Version 11 Application Programming APIs*.

Maintaining database integrity (IMS batch, BMP, and IMS online regions)

IMS uses these DL/I calls to back out database updates: ROLB, ROLL, ROLS, SETS, and SETU.

The ROLB and ROLS calls can back out the database updates or cancel the output messages that the program has created since the program's most recent commit point. A ROLL call backs out the database updates and cancels any non-express output messages the program has created since the last commit point. It also deletes the current input message. SETS allows multiple intermediate backout points to be noted during application program processing. SETU operates like SETS except that it is not rejected by unsupported PCBs in the PSB. If your program issues a subsequent ROLS call specifying one of these points, database updates and message activity performed since that point are backed out.

CICS online programs with DBCTL can use the ROLS and SETS or SETU DL/I calls to back out database changes to a previous commit point or to an intermediate backout point.

Backing out to a prior commit point: ROLL, ROLB, and ROLS

When a program determines that some of its processing is invalid, some calls enable the program to remove the effects of its incorrect processing. These are the Roll Back calls: ROLL, ROLS using a DB PCB (or ROLS without an I/O area or token), and ROLB.

When you issue one of these calls, IMS:

- Backs out the database updates that the program has made since the program's most recent commit point.
- Cancels the non-express output messages that the program has created since the program's most recent commit point.

The main difference between these calls is that ROLB returns control to the application program after backing out updates and canceling output messages, ROLS does not return control to the application program, and ROLL terminates the program with an abend code of U0778. ROLB can return the first message segment to the program since the most recent commit point, but ROLL and ROLS cannot.

The ROLL and ROLB calls, and the ROLS call without a specified token, are valid when the PSB contains PCBs for GSAM data sets. However, segments inserted in the GSAM data sets since the last commit point are not backed out by these calls. An extended checkpoint-restart can be used to reposition the GSAM data sets when restarting.

You can use a ROLS call either to back out to the prior commit point or to back out to an intermediate backout point that was established by a prior SETS call. This section refers only to the form of the ROLS call that backs out to the prior commit point. For information about the other form of ROLS, see 'Backing out to an intermediate backout point: SETS, SETU, and ROLS'.

The table below summarizes the similarities and the differences between the ROLB, ROLL, and ROLS calls.

Table 42. Comparison of ROLB, ROLL, and ROLS.

Actions Taken:	**ROLB**	**ROLL**	**ROLS**
Back out database updates since the last commit point.	X	X	X
Cancel output messages created since the last commit point.	X[1]	X[1]	X[1]
Delete from the queue the message in process. Previous messages (if any) processed since the last commit point are returned to the queue to be reprocessed.		X	
Return the first segment of the first input message issued since the most recent commit point.	X[2]		
U3303 abnormal termination. Returns the processed input messages to the message queue.			X[3]
U0778 abnormal termination. No dump.		X	
No abend. Program continues processing.	X		

Notes:

1. ROLB, ROLL, or ROLS calls cancel output messages that are sent with an express PCB unless the program issued a PURG. For example, if the program issues the call sequence that follows, MSG1 would be sent to its destination because PURG tells IMS that MSG1 is complete and the I/O area now contains the first segment of the next message (which in this example is MSG2). MSG2, however, would be canceled.

   ```
   ISRT   EXPRESS PCB, MSG1
   PURG   EXPRESS PCB, MSG2
   ROLB   I/O PCB
   ```

   Because IMS has the complete message (MSG1) and because an express PCB is being used, the message can be sent before a commit point.
2. Returned only if you supply the address of an I/O area as one of the call parameters.
3. The transaction is suspended and requeued for subsequent processing.

ROLL call

A ROLL call backs out the database updates and cancels any non-express output messages the program has created since the last commit point. It also deletes the current input message. Any other input messages that were processed since the last commit point are returned to the queue to be reprocessed. IMS then terminates

the program with an abend code U0778. This type of abnormal termination terminates the program without a storage dump.

When you issue a ROLL call, the only parameter you supply is the call function, ROLL.

You can use the ROLL call in a batch program. If your system log is on DASD, and if dynamic backout has been specified through the use of the BKO execution parameter, database changes made since the last commit point will be backed out; otherwise they will not. One reason for issuing ROLL in a batch program is for compatibility.

After backout is complete, the original transaction is discarded if it can be, and it is not re-executed. IMS issues the APPC/MVS verb, ATBCMTP TYPE(ABEND), specifying the TPI to notify remote transaction programs. Issuing the APPC/MVS verb causes all active conversations (including any that are spawned by the application program) to be DEALLOCATED TYP(ABEND_SVC).

ROLB call

The advantage of using a ROLB call is that IMS returns control to the program after executing a ROLB call, so the program can continue processing. The parameters for the ROLB call are:

- The call function, ROLB
- The name of the I/O PCB or AIB

The total effect of the ROLB call depends on the type of IMS application program that issued it.

- For current IMS application programs:

  After IMS backout is complete, the original transaction is represented to the IMS application program. Any resources that cannot be rolled back by IMS are ignored; for example, output that is sent to an express alternate PCB and a PURG call that is issued before the ROLB call.
- For modified IMS application programs:

  The same consideration for the current IMS application program applies. The application program must notify any spawned conversations that a ROLB was issued.
- For CPI-C driven IMS application programs:

  Only IMS resources are affected. All database changes are backed out. Any messages that are inserted to non-express alternate PCBs are discarded. Also, any messages that are inserted to express PCBs that have not had a PURG call are discarded. The application program must notify the originating remote program and any spawned conversations that a ROLB call was issued.

MPPs and transaction-oriented BMPs

If the program supplies the address of an I/O area as one of the ROLB parameters, the ROLB call acts as a message retrieval call and returns the first segment of the first input message issued since the most recent commit point. This is true only if the program has issued a GU call to the message queue since the last commit point; it if has not, it was not processing a message when it issued the ROLB call.

If the program issues GN call to the message queue after issuing a ROLB call, IMS returns the next segment of the message that was being processed when the ROLB call was issued. If no more segments exist for that message, IMS returns a QD status code.

If the program issues a GU call to the message queue after the ROLB call, IMS returns the first segment of the next message to the application program. If no more messages exist on the message queue for the program to process, IMS returns a QC status code.

If you include the I/O area parameter, but you have not issued a successful GU call to the message queue since the last commit point, IMS returns a QE status code to your program.

If you do not include the address of an I/O area in the ROLB call, IMS does the same thing for you. If the program has issued a successful GU call in the commit interval and then issues a GN call, IMS returns a QD status code. If the program issues a GU call after the ROLB call, IMS returns the first segment of the next message or a QC status code, if no more messages exist for the program.

If you have not issued a successful GU call since the last commit point, and you do not include an I/O area parameter on the ROLB call, IMS backs out the database updates and cancels the output messages that were created since the last commit point.

Batch programs

If your system log is on DASD, and if dynamic backout has been specified through the use of the BKO execution parameter, you can use the ROLB call in a batch program. The ROLB call does not process messages as it does for MPPs; it backs out the database updates made since the last commit point and returns control to your program. You cannot specify the address of an I/O area as one of the parameters on the call; if you do, an AD status code is returned to your program. You must, however, have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB.

ROLS call

You can use the ROLS call in two ways to back out to the prior commit point and return the processed input messages to IMS for later reprocessing:

- Have your program issue the ROLS call using the I/O PCB but without an I/O area or token in the call. The parameters for this form of the ROLS call are:

  The call function, ROLS

  The name of the I/O PCB or AIB
- Have your program issue the ROLS call using a database PCB that has received one of the data-unavailable status codes. This has the same result as if unavailable data were encountered and the INIT call was not issued. A ROLS call must be the next call for that PCB. Intervening calls using other PCBs are permitted.

On a ROLS call with a TOKEN, message queue repositioning can occur for all non-express messages, including all messages processed by IMS. The processing uses APPC/MVS calls, and includes the initial message segments. The original input transaction can be represented to the IMS application program. Input and output positioning is determined by the SETS call. This positioning applies to

current and modified IMS application programs but does not apply to CPI-C driven IMS programs. The IMS application program must notify all remote transaction programs of the ROLS.

On a ROLS call without a TOKEN, IMS issues the APPC/MVS verb, ATBCMTP TYPE(ABEND), specifying the TPI. Issuing this verb causes all conversations associated with the application program to be DEALLOCATED TYPE(ABEND_SVC). If the original transaction is entered from an LU 6.2 device and IMS receives the message from APPC/MVS, a discardable transaction is discarded rather than being placed on the suspend queue like a non-discardable transaction.

The parameters for this form of the ROLS call are:

- The call function, ROLS
- The name of the DB PCB that received the BA or BB status code

In both of the these parameters, the ROLS call causes a U3303 abnormal termination and does not return control to the application program. IMS keeps the input message for future processing.

Related concepts:

Administering APPC/IMS and LU 6.2 devices (Communications and Connections)

Related reference:

Program Specification Block (PSB) Generation utility (System Utilities)

ROLB call (Application Programming APIs)

Backing out to an intermediate backout point: SETS, SETU, and ROLS

You can use a ROLS call either to back out to an intermediate backout point that was established by a prior SETS or SETU call, or to back out to the prior commit point.

The ROLS call that backs out to an intermediate point backs out only DL/I changes. This version of the ROLS call does not affect CICS changes that use CICS file control or CICS transient data.

The SETS and ROLS calls set intermediate backout points within the call processing of the application program and then backout database changes to any of these points. Up to nine intermediate backout points can be set. The SETS call specifies a token for each point. IMS then associates this token with the current processing point. A subsequent ROLS call using the same token backs out all database changes and discards all non-express messages that were performed after the SETS call with the same token. The following figure shows how the SETS and ROLS calls work together.

In addition, to assist the application program in managing other variables that it may want to reestablish after a ROLS call, user data can be included in the I/O area of the SETS call. This data is then returned when the ROLS call is issued with the same token.

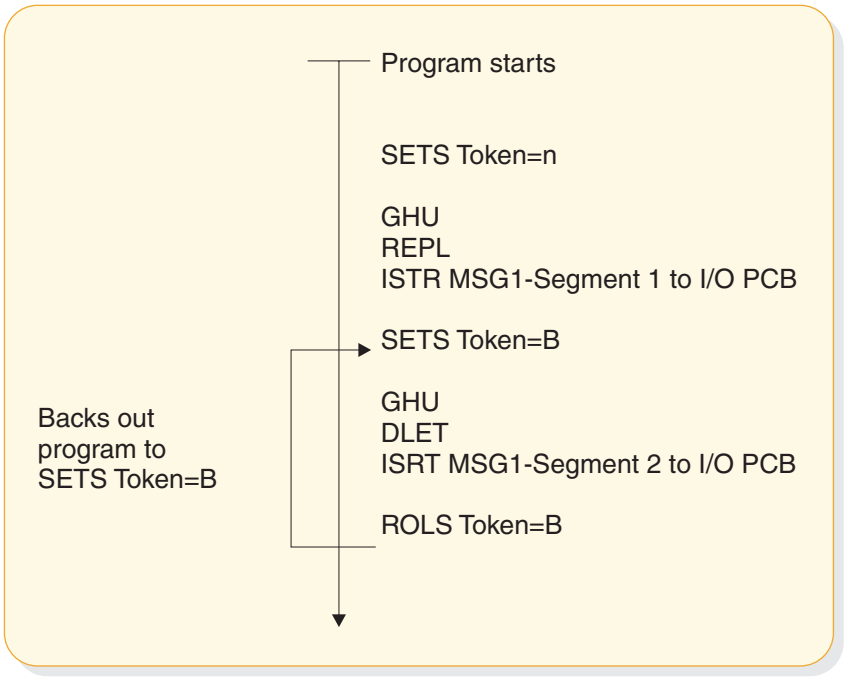


Figure 59. SETS and ROLS calls working together

SETS and SETU calls

The SETS call sets up to nine intermediate backout points or cancels all existing backout points. With the SETS call, you can back out pieces of work. If the necessary data to complete one piece of work is unavailable, you can complete a different piece of work and then return to the former piece.

To set an intermediate backout point, issue the call using the I/O PCB, and include an I/O area and a token. The I/O area has the format *LLZZuser-data*, where *LL* is the length of the data in the I/O area including the length of the *LLZZ* portion. The *ZZ* field must contain binary zeros. The data in the I/O area is returned to the application program on the related ROLS call. If you do not want to save some of the data that is to be returned on the ROLS call, set the *LL* that defines the length of the I/O area to 4.

For PLITDLI, you must define the *LL* field as a fullword rather than a halfword, as it is for the other languages. The content of the *LL* field for PLITDLI is consistent with the I/O area for other calls using the *LLZZ* format. The content is the total length of the area, including the length of the 4-byte *LL* field, minus 2.

A 4-byte token associated with the current processing point is also required. This token can be a new token for this program execution, or it can match a token that was issued by a preceding SETS call. If the token is new, no preceding SETS calls are canceled. If the token matches the token of a preceding SETS call, the current SETS call assumes that position. In this case, all SETS calls that were issued subsequent to the SETS call with the matching token are canceled.

The parameters for this form of the SETS call are:

- The call function, SETS
- The name of the I/O PCB or AIB
- The name of the I/O area containing the user data

- The name of an area containing the token

For the SETS call format, see the topic "SETS/SETU Call" in *IMS Version 11 Application Programming APIs*.

To cancel all previous backout points, the call is issued using the I/O PCB but does not include an I/O area or a token. When an I/O area is not included in the call, all intermediate backout points that were set by prior SETS calls are canceled.

The parameters for this form of the SETS call are:

- The call function, SETS
- The name of the I/O PCB or AIB

Because it is not possible to back out committed data, commit-point processing causes all outstanding SETS to be canceled.

If PCBs for DEDB, MSDB, and GSAM organizations are in the PSB, or if the program accesses an attached subsystem, a partial backout is not possible. In that case, the SETS call is rejected with an SC status code. If the SETU call is used instead, it is not rejected because of unsupported PCBs, but will return an SC status code as a warning that the PSB contains unsupported PCBs and that the function is not applicable to these unsupported PCBs.

Related reading: For status codes that are returned after the SETS call and the explanations of those status codes and the response required, see *IMS Version 11 Application Programming APIs*.

ROLS

The ROLS call backs out database changes to a processing point set by a previous SETS or SETU call, or to the prior commit point. The ROLS call then returns the processed input messages to the message queue.

To back out database changes and message activity that have occurred since a prior SETS call, issue the ROLS call using the I/O PCB, and specify an I/O area and token in the call. If the token does not match a token that was set by a preceding SETS call, an error status is returned. If the token matches the token of a preceding SETS call, the database updates made since this corresponding SETS call are backed out, and all non-express messages that were inserted since the corresponding SETS are discarded. SETS that are issued as part of processing that was backed out are canceled. The existing database positions for all supported PCBs are reset.

If a ROLS call is in response to a SETU call, and if there are unsupported PCBs (DEDB, MSDB, or GSAM) in the PSB, the position of the PCBs is not affected. The token specified by the ROLS call can be set by either a SETS or SETU call. If no unsupported PCBs exist in the PSB, and if the program has not used an attached subsystem, the function of the ROLS call is the same regardless of whether the token was set by a SETS or SETU call.

If the ROLS call is in response to a SETS call, and if unsupported PCBs exist in the PSB or the program used an attached subsystem when the preceding SETS call was issued, the SETS call is rejected with an SC status code. The subsequent ROLS call is either rejected with an RC status code, indicating unsupported options, or it is rejected with an RA status code, indicating that a matching token that was set by a preceding successful SETS call does not exist.

If the ROLS call is in response to a SETU call, the call is not rejected because of unsupported options. If unsupported PCBs exist in the PSB, this is not reflected with a status code on the ROLS call. If the program is using an attached subsystem, the ROLS call is processed, but an RC status is returned as a warning indicating that if changes were made using the attached subsystem, those changes were not backed out.

The parameters for this form of the ROLS call are:

- The call function, ROLS
- The name of the I/O PCB or AIB
- The name of the I/O area to receive the user data
- The name of an area containing the 4-byte token

Related reading: For status codes that are returned after the ROLS call and the explanations of those status codes and the response require, see *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes*.

Related concepts:

"Backing out to a prior commit point: ROLL, ROLB, and ROLS calls" on page 431

Reserving segments for the exclusive use of your program

You may want to reserve a segment and prohibit other programs from updating the segment while you are using it. To some extent, IMS does this for you through resource lock management. The Q command code lets you reserve segments in a different way.

Restriction: The Q command code is not supported for MSDB organizations or for a secondary index that is processed as a database.

Resource lock management and the Q command code both reserve segments for your program's use, but they work differently and are independent of each other. To understand how and when to use the Q command code and the DEQ call, you must understand resource lock management.

The function of resource lock management is to prevent one program from accessing data that another program has altered until the altering program reaches a commit point. Therefore, you know that if you have altered a segment, no other program (except those using the GO processing option) can access that segment until your program reaches a commit point. For database organizations that support the Q command code, if the PCB processing option allows updates and the PCB holds position in a database record, no other program can access the database record.

The Q command code allows you to prevent other programs from updating a segment that you have accessed, even when the PCB that accessed the segment moves to another database record.

Related reading: For more information on the Q command code, see the topic "Q command code" in *IMS Version 11 Application Programming APIs*.

Chapter 17. Secondary indexing and logical relationships

Secondary indexing and logical relationships are techniques that can change your application program's view of the data. The DBA makes the decision about whether to use these options.

Examples of when you use these techniques are:

- If an application program must access a segment type in a sequence other than the sequence specified by the key field, secondary indexing can be used. Secondary indexing also can change the application program's access to or view of the data based on a condition in a dependent segment.
- If an application program requires a logical structure that contains segments from different databases, logical relationships are used.

Related concepts:

"SSA guidelines" on page 185

How secondary indexing affects your program

One instance of using a secondary index occurs when an application program needs to select database records in a sequence other than that defined by the root key.

IMS stores root segments in the sequence of their key fields. A program that accesses root segments out of the order of their key fields cannot operate efficiently.

You can index any field in a segment by defining an XDFLD statement for the field in the DBD for the database. If the Get call is not qualified on the key but uses some other field, IMS must search all the database records to find the correct record. With secondary indexing, IMS can go directly to a record based on a field value that is not in the key field.

For more information about secondary indexes and examples, see *IMS Version 11 Database Administration*.

SSAs with secondary indexes

If your program uses a secondary index, you can use the name of an indexed field in your SSAs. When you do this, IMS goes directly to the secondary index and finds the pointer segment with the value you specify. Then IMS locates the segment that the index segment points to in the primary database and returns the segment to your program.

To use an indexed field name in the SSA, follow these guidelines:

- Define the indexed field, using the XDFLD statement, in the DBD for the primary database during DBD generation.
- Use the name that was given on the XDFLD statement as the field name in the qualification statement.
- Specify the secondary index as the processing sequence during PSB generation. Do this by specifying the name of the secondary index database on the PROCSEQ parameter on the PCB during PSB generation.

© Copyright IBM Corp. 1974, 2015

If you modify the XDFLD of the indexed segment (using the REPL call), you lose any parentage that you had established before issuing the REPL call. The key feedback area is no longer valid after a successful REPL call.

For example, to index the PATIENT segment on the NAME field, the segment must have been defined on the XDFLD statement in the DBD for the medical database. If the name of the secondary index database is INDEX, you specify PROCSEQ=INDEX in the PCB. To issue a qualification that identifies a PATIENT by the NAME field instead of by PATNO, use the name that you specified on the XDFLD statement. If the name of the XDFLD is XNAME, use XNAME in the SSA, as follows:

In the DBD:

```
XDFLD NAME=XNAME
```

In the PSB:

```
PROCSEQ=INDEX
```

In the program:

```
GU PATIENTb(XNAMEbbb=bJBBROKEbbb)
```

Multiple qualification statements with secondary indexes

When you qualify a call using the name of an indexed field, you can include multiple qualification statements.

You can use two AND operators to connect the qualification statements:

*** or &** When used with secondary indexing, this AND is called the dependent AND. To satisfy the call, IMS scans the index once and searches for one pointer segment in the index that satisfies both qualification statements.

This is called the independent AND. You use it only with secondary indexing. When you use the independent AND to satisfy the call, IMS scans the index twice and searches for two or more different pointer segments in the index that point to the same target segment.

The distinction between the two ANDs applies only when the indexed field (the one defined as XDFLD in the DBD) is used in all qualifications. If one of the qualification statements uses another field, both ANDs work like the dependent AND.

The next two sections give examples of the dependent and independent AND. Although the examples show only two qualification statements in the SSA, you can use more than two. No set limit exists for the number of qualification statements you can include in an SSA, but a limit on the maximum size of the SSA does exist. You specify this size on the SSASIZE parameter of the PSBGEN statement. For information on this parameter, see *IMS Version 11 System Utilities*.

The dependent AND

When you use the dependent AND, IMS scans the index only once. To satisfy the call, it must find **one** pointer segment that satisfies both qualification statements.

For example, suppose you want to list patients whose bills are between $500 and $1000. To do this, you index the PATIENT segment on the BILLING segment, and specify that you want IMS to use the secondary index as the processing sequence. The following figure shows the three secondary indexing segments.

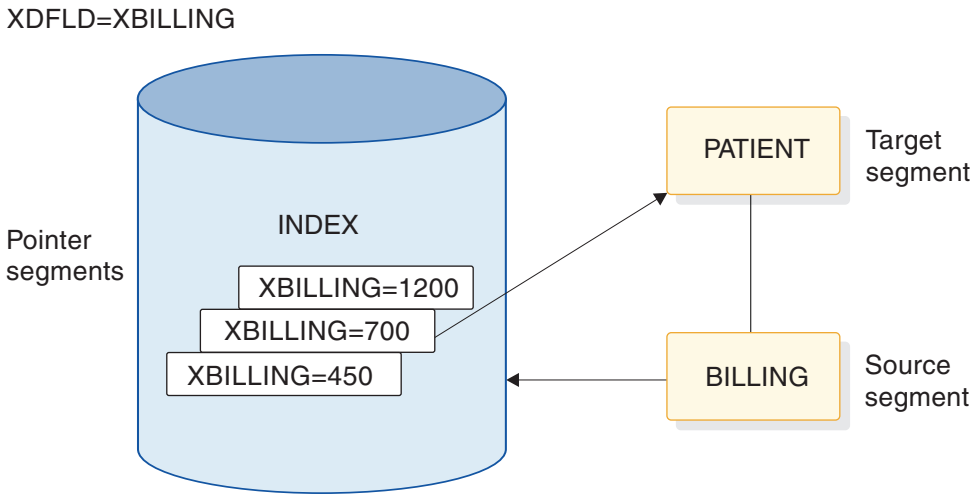


Figure 60. Example of using the dependent AND

You then use this call:

```
GU   PATIENT (XBILLING>=00500*XBILLING<=01000)
```

To satisfy this call, IMS searches for one pointer segment with a value between 500 and 1000. IMS returns the PATIENT segment that is pointed to by that segment.

The independent AND

For example, suppose you want a list of the patients who have had both tonsillitis and strep throat. To get this information, you index the PATIENT segment on the ILLNAME field in the ILLNESS segment, and specify that you want IMS to use the secondary index as the processing sequence. In this example, you retrieve the PARENT segments based on a dependent's (the ILLNESS segment's) qualification. The following figure shows the four secondary indexing segments.

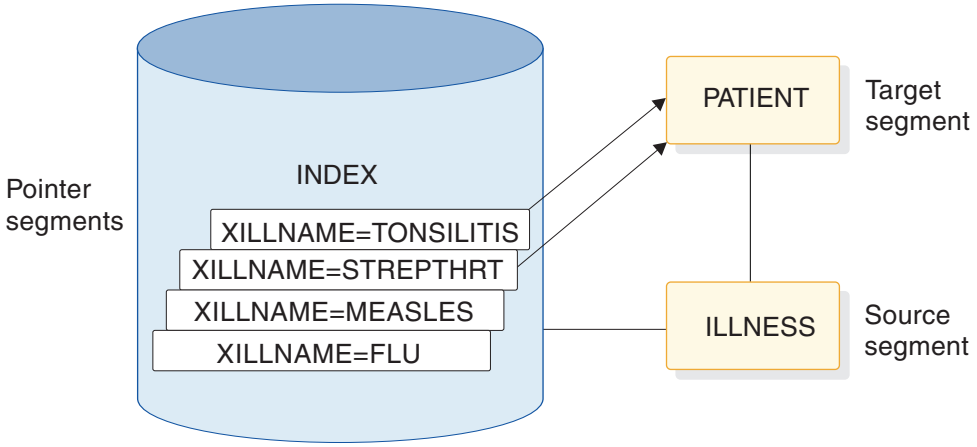


Figure 61. Example of using the independent AND

You want IMS to find two pointer segments in the index that point to the same PATIENT segment, one with ILLNAME equal to TONSILLITIS and one with ILLNAME equal to STREPTHRT. Use this call:

```
GU   PATIENT⌂
(XILLNAME=TONSILITIS#XILLNAME=⌂
STREPTHRT)
```

This call retrieves the first PATIENT segment with ILLNESS segments of strep throat and tonsillitis. When you issue the call, IMS searches for an index entry for tonsillitis. Then it searches for an index entry for strep throat that points to the same PATIENT segment.

When you use the independent AND with GN and GNP calls, a special situation can occur. If you repeat a GN or a GNP call using the same qualification, it is possible for IMS to return the same segment to your program more than once. You can check to find out whether IMS has already returned a segment to you by checking the key feedback area.

If you continue issuing a GN call until you receive a not-found (GE) status code, IMS returns a segment occurrence once for each independent AND group. When IMS returns a segment that is identical to one that was already returned, the PCB key feedback area is different.

Related concepts:

"Multiple qualification statements" on page 186

DL/I returns with secondary indexes

The term "key of the pointer segment" refers to the key as perceived by the application program. That is, the key does not include subsequent fields. IMS places this key in the position where the root key would be located if you had not used a secondary index—in the left-most bytes of the key feedback area.

The PATIENT segment that IMS returns to the application program's I/O area looks just as it would if you had not used secondary indexing. The key feedback area, however, contains something different. The concatenated key that IMS returns is the same, except that, instead of giving you the key for the segment you requested (the key for the PATIENT segment), IMS gives you the search portion of the key of the secondary index (the key for the segment in the INDEX database).

If you try to insert or replace a segment that contains a secondary index source field that is a duplicate of one that is already reflected in the secondary index, IMS returns an NI status code. An NI status code is returned only for batch programs that log to direct-access storage. Otherwise, the application program is abnormally terminated. You can avoid having your program terminated by making sure a duplicate index source field does not exist. Before inserting a segment, try to retrieve the segment using the secondary index source field as qualification.

Status codes for secondary indexes

If a secondary index is defined for a segment and if the definition specifies a unique key for the secondary index (most secondary indexes allow duplicate keys), your application program might receive the NI status code in addition to regular status codes.

This status code can be received for a PCB that either uses or does not use the secondary index as a processing sequence. See *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes* for additional information about the NI status code.

Processing segments in logical relationships

Sometimes an application program needs to process a hierarchy that is made up of segments that already exist in two or more separate database hierarchies. Logical relationships make it possible to establish hierarchic relationships between these segments. When you use logical relationships, the result is a new hierarchy—one that does not exist in physical storage but that can be processed by application programs as though it does exist. This type of hierarchy is called a logical structure.

One advantage of using logical relationships is that programs can access the data as though it exists in more than one hierarchy, even though it is only stored in one place. When two application programs need to access the same segment through different paths, an alternative to using logical relationships is to store the segment in both hierarchies. The problem with this approach is that you must update the data in two places to keep it current.

Processing segments in logical relationships is not very different from processing other segments. The following examples are taken from a scenario for an inventory application program that processes data in a purchasing database, but which also needs access to a segment in a patient database.

For example, the hierarchy that an inventory application program needs to process contains four segment types:

- An ITEM segment containing the name and an identification number of a medication that is used at a medical clinic
- A VENDOR segment that contains the name and address of the vendor who supplies the item
- A SHIPMENT segment that contains information such as quantity and date for each shipment of the item that the clinic receives
- A DISBURSE segment that contains information about the disbursement of the item at the clinic, such as the quantity, the date, and the doctor who prescribed it

The TREATMNT segment in the medical database contains the same information that the inventory application program needs to process in the DISBURSE segment. Rather than store this information in both hierarchies, you can store the information in the TREATMNT segment, and define a logical relationship between the DISBURSE segment in the item hierarchy and the TREATMNT segment in the patient hierarchy. Doing this makes it possible to process the TREATMNT segment through the item hierarchy as though it is a child of SHIPMENT. DISBURSE then has two parents: SHIPMENT is DISBURSE's physical parent, and TREATMNT is DISBURSE's logical parent.

Three segments are involved in this logical relationship: DISBURSE, SHIPMENT, and TREATMNT. The following figure shows the item hierarchy on the right. The DISBURSE segment points to the TREATMNT segment in the patient hierarchy shown on the left. (The patient hierarchy is part of the medical database.)

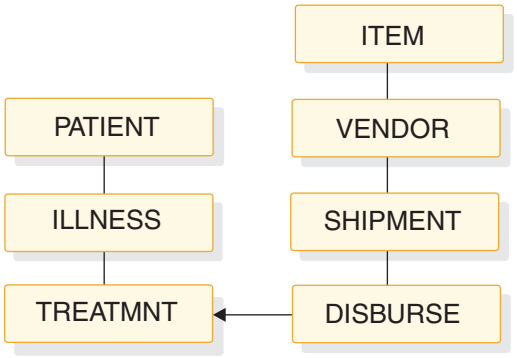


Figure 62. Patient and item hierarchies

Three types of segments are found in a logical relationship:

- TREATMNT is called the logical parent segment. It is a physical dependent of ILLNESS, but it can be processed through the item hierarchy because a path is established by the logical child segment DISBURSE. The logical parent segment can be accessed through both hierarchies, but it is stored in only one place.
- SHIPMENT is called a physical parent segment. The physical parent is the parent of the logical child in the physical database hierarchy.
- DISBURSE is called a logical child segment. It establishes a path to the TREATMNT segment in the PATIENT hierarchy from the SHIPMENT segment in the ITEM hierarchy.

Because a logical child segment points to its logical parent, two paths exist through which a program can access the logical parent segment:

- When a program accesses the logical parent segment through the physical path, it reaches this logical parent segment through the segment's physical parent. Accessing the TREATMNT segment through ILLNESS is accessing the logical parent segment through its physical path.
- When a program accesses the logical parent segment through the logical path, it reaches this logical parent segment through the segment's logical child. Accessing the TREATMNT segment through SHIPMENT is accessing the logical parent segment through its logical path.

When a logical parent segment is accessed through the logical child, the logical child is concatenated with both the data from its logical parent segment and any data the user has chosen to associate with this pairing (intersection data) in a single segment I/O area, like this:

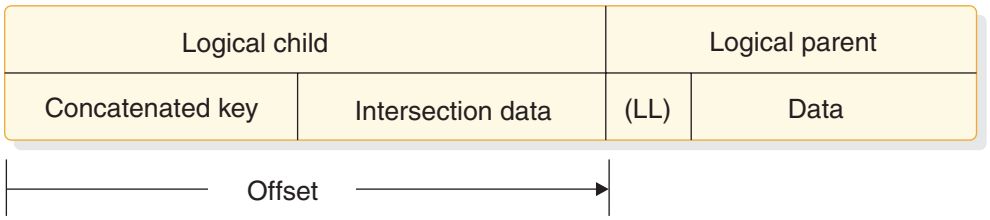


Figure 63. Concatenated segment

LL is the length field of the logical parent if this segment is a variable-length segment.

How logical relationships affect your programming

The calls you issue to process segments in logical relationships are the same calls that you use to process other segments. However, the processing is different depending on how the logical segment looks in your I/O area, what the DB PCB mask contains after a retrieve call, and how you can replace, delete, and insert physical and logical parent segments.

Because it is possible to access segments in logical relationships through the logical path or the physical path, the segments must be protected from being updated by unauthorized programs.

When DBAs define logical relationships, they define a set of rules that determine how the segments can be deleted, replaced, and inserted. Defining these rules is a database design decision. If your program processes segments in logical relationships, the DBA (or the person at your installation responsible for database design) should tell you:

- What segments look like in your I/O area when you retrieve them
- Whether your program is allowed to update and insert segments
- What to do if you receive a DX, IX, or RX status code

The requirements for inserting a logical child segment are:

- In load mode, the logical child can be inserted only under its physical parent. You do not supply the logical parent in the I/O area.
- In update mode, the format of the logical child is different, depending on whether it is accessed from its physical parent or from its logical parent.
  - If accessed from its physical parent, the logical child's format is the concatenated key of the logical parent followed by intersection data.
  - If accessed from its logical parent, the logical child's format is the concatenated key of the physical parent, followed by intersection data.
- The logical child can be inserted or replaced, depending on the insert rule for the logical or physical parent. Unless the insert rule of the logical or physical parent is PHYSICAL, the logical or physical parent must be supplied in the I/O area following the logical child.

Related concepts:

"Multiple qualification statements" on page 186

Status codes for logical relationships

These status codes apply specifically to segments that are involved in logical relationships.

These are not all of the status codes that you can receive when processing a logical child segment or a physical or logical parent. If you receive one of these status codes, it means that you are trying to update the database in a way that you are not allowed to. Check with the DBA or person responsible for implementing logical relationships at your installation to find out what the problem is.

DX IMS did not delete the segment because the physical delete rule was violated. If the segment is a logical parent, it still has active logical children. If the segment is a logical child, it has not been deleted through its logical path.

IX You tried to insert either a logical child segment or a concatenated segment. If it was a logical child segment, the corresponding logical or physical parent segment does not exist. If it was a concatenated segment,

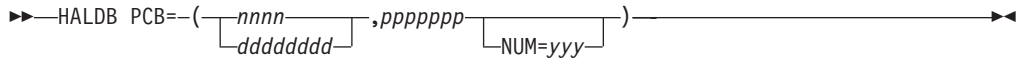
either the insert rule was physical and the logical or physical parent does not exist, or the insert rule is virtual and the key of the logical or physical parent in the I/O area does not match the concatenated key of the logical or physical parent.

RX The physical replace rule has been violated. The physical replace rule was specified for the destination parent, and an attempt was made to change its data. When a destination parent has the physical replace rule, it can be replaced only through the physical path.

Chapter 18. HALDB selective partition processing

You can restrict the processing of DL/I calls to a single HALDB partition or a range of HALDB partitions by using a DD statement with the ddname DFSHALDB to pass control statements. DFS HALDB must be provided in the JCL of the batch job, the BMP (Batch Message Processing dependent online region), or the JBP (Java Batch Processing dependent online region).

Control Statements for HALDB selective partition processing



Each HALDB control statement must have a PCB keyword that contains the required parameters. The required parameters for an individual control statement must be on one line; no continuation is allowed. The input can consist of multiple HALDB control statements. There should be no duplication of DB PCB numbers. In the event of a duplication, the control statement that has been read the most recently overrides the previous statement.

Any HALDB control statement that is syntactically correct results in an entry within a table. The maximum number of entries in the table is 20. All subsequent statements that are read, even though syntactically correct, are ignored and result in a U0201 abend, unless a statement is a duplicate of an entry that is already in the table.

Parameter descriptions for HALDB selective partition processing

nnnn
The DB PCB number as the relative number of the DB PCB defined in the PSB.

dddddddd
The DB PCB label or name.

ppppppp
The partition name. This parameter is required.

NUM=*yyy*
The range of consecutive partitions that this PCB is restricted to using, starting with the named partition. The range of consecutive partitions is defined as the partition selection order, which is the next partition selected starting from the target partition named in the DFSHALDB statement. The next partition is determined using either the high keys defined for the HALDB or the processing order defined by the partition selection exit. This parameter is optional.

The following examples show how to use HALDB selective partition processing statements.

DFSHALDB for single partition restriction

```
HALDB PCB=(4,POHIDKA)
HALDB PCB=(PCBNUM2,POHIDJA)
```

© Copyright IBM Corp. 1974, 2015

DFSHALDB for range partition restriction

```
HALDB PCB=(3,PVHDJ5A,NUM=4)
HALDB PCB=(PCBNUM7,PVHDJ5B,NUM=3)
```

DFSHALDB for independent processing of partitions

```
PRINT NOGEN
PCB TYPE=DB,DBDNAME=G1CSTP,PROCOPT=A,KEYLEN=100,PCBNAME=XXCSTP
SENSEG NAME=CUSTOMER
SENSEG NAME=DISTRICT,PARENT=CUSTOMER
SENSEG NAME=CUSTLOCN,PARENT=CUSTOMER
SENSEG NAME=ADDRLINE,PARENT=CUSTLOCN
SENSEG NAME=CUSTORDN,PARENT=CUSTLOCN
SENSEG NAME=CUSTINVN,PARENT=CUSTOMER
SENSEG NAME=PAYMENTS,PARENT=CUSTOMER
SENSEG NAME=ADJUSTMT,PARENT=CUSTOMER

PCB TYPE=DB,DBDNAME=G1CSTP,PROCOPT=A,KEYLEN=100,PCBNAME=IICSTP,
              PROCSEQ=CSTCY02
SENSEG NAME=CUSTOMER


PSBGEN LANG=ASSEM,PSBNAME=G1ACSTP,CMPAT=YES
END
```

Report generated for HALDB selective partition processing

When you use HALDB selective partition processing, a report called "HALDB Selective Partition Processing" is generated in the SYSHALDB data set. This report shows the control statements that have been issued and the reason for accepting or rejecting each statement. Control statements that have been validated and accepted are shown as "Syntactically correct." Other messages that might appear for syntactically correct statements, and their accompanying messages, are shown in the following table:

Table 43. Messages provided in the report generated for HALDB selective partition processing

Message	Explanation
Duplicate, overrides previous statement	A HALDB statement for the same PCB was already found. The current statement overrides the previous HALDB statement.
Ignored, number of valid statements exceeds 20	More than 20 HALDB statements were provided, but only 20 statements are allowed. Reduce the number of HALDB statements to 20 or fewer, and run the job again. This message results in an abend U0201.
NUM parameter must be non-zero numeric	The partition range specified in the NUM keyword must be a non-zero value from 1 to 999.
NUM value exceeds three digits	The partition range specified in the NUM keyword must be a non-zero value from 1 to 999.
An equal sign must follow NUM keyword	An equal sign must follow the NUM keyword in the HALDB statement. Add an equal sign to the HALDB statement.

Table 43. Messages provided in the report generated for HALDB selective partition processing (continued)

Message	Explanation
The NUM keyword is missing	A comma was found after the partition name, but the NUM keyword was not present. Either verify the syntax of the positional parameters in the HALDB statement, or add the NUM keyword and the range of partitions for the restriction.
NUM parameter is missing	The NUM keyword was found, but the NUM parameter value was not present. Either verify the syntax of the positional parameters in the HALDB statement, or add the NUM keyword and the range of partitions for the restriction.

For HALDB control statements that are not syntactically correct (statements that are processed and rejected), the messages and explanations that are issued are shown in the following table:

Table 44. Messages provided in the report generated for syntactically incorrect HALDB statements

Message	Explanation
No HALDB statement type	The DFSHALDB data set did not contain a HALDB statement. Add a HALDB statement to prevent this error.
A space must follow HALDB statement type	The HALDB statement requires a space after HALDB and before the PCB keyword.
PCB keyword missing	The required keyword PCB was not found. The PCB keyword must be present to process the HALDB statement successfully.
Equal sign must follow PCB keyword	An equal sign did not follow the PCB keyword. The equal sign must follow the PCB keyword to process the HALDB statement successfully.
Open parenthesis must follow equal sign	An open parenthesis did not follow PCB=. The open parenthesis must follow the PCB= to process the HALDB statement successfully.
Second parameter may be missing	The HALDB partition must be provided. Either add the partition name, or verify that the syntax of the positional parameters is correct.
First parameter exceeds four digits	The DB PCB number cannot exceed a four-digit value. Change the DB PCB number to the correct DB PCB number.
Delimiter is not a comma	A comma is missing between parameter values. The comma is used as a delimiter for the positional parameters. Either add the comma, or verify that the syntax of the positional parameters is correct.

Table 44. Messages provided in the report generated for syntactically incorrect HALDB statements (continued)

Message	Explanation
Partition name must start with an alpha	The HALDB partition name must begin with a alphabetic character. Add the partition name or verify the syntax of the positional parameters is correct.
Delimiter is not a close parenthesis	A closing parenthesis is missing from the HALDB statement. Add a closing parenthesis around the PCB parameters.
Partition name exceeds seven characters	The HALDB partition name must be seven or fewer characters. Either add the partition name, or verify that the syntax of the positional parameters is correct.
Invalid character in partition name	The HALDB partition name contains an invalid character. Either add the partition name, or verify that the syntax of the positional parameters is correct.
Statement contains all spaces	The HALDB statement is missing. Add a valid HALDB statement.
Invalid statement input	A HALDB statement was found, but it does not appear to be complete. Verify the syntax of the HALDB statement and the positional parameters specified.
Space must follow close parenthesis	A space must follow the closing parenthesis. Add a space after the closing parenthesis.
First parameter missing	The PCB number or label is missing. Either add the PCB name or label, or verify that the syntax of the positional parameters is correct.
Comma and part name missing	Only the PCB number or label was provided in the HALDB statement. Either add the partition name, or verify that the syntax of the positional parameters is correct.
Partition name is missing	The HALDB partition name must be provided in the HALDB statement. Either add the partition name, or verify that the syntax of the positional parameters is correct.
Partition name starts with numeric	The HALDB partition name must begin with an alphabetic character. Either add the partition name, or verify that the syntax of the positional parameters is correct.
First parameter must not be zero	The PCB number must be a non-zero number. Add a non-zero number for the DB PCB number.
Comment statement	An asterisk was found in column one of the HALDB statement. This statement was skipped and considered a comment.

After all of the statements are validated, the job abnormally terminates with an abend code of U0201.

Chapter 19. Processing GSAM databases

GSAM databases are available to application programs that can run as batch programs in batch message processing (BMP) regions, transaction-oriented BMPs, or Java batch processing (JBP) regions.

If your application program accesses GSAM databases, as you design your program consider that:

- An IMS program can retrieve records and add records to the end of the GSAM database, but the program cannot delete or replace records in the database.
- You use separate calls to access GSAM databases. (Additional checkpoint and restart considerations are involved in using GSAM.)
- Your program must use symbolic `CHKP` and `XRST` calls if it uses GSAM. Basic `CHKP` calls cannot checkpoint GSAM databases.
- When an IMS program uses a GSAM data set, the program treats a GSAM data set like a sequential non-hierarchic database. The z/OS access methods that GSAM can use are BSAM on direct access, unit record, and tape devices; and VSAM on direct-access storage. VSAM data sets must be non-keyed, non-indexed, entry-sequenced data sets (ESDS) and must reside on DASD. VSAM does not support temporary, SYSIN, SYSOUT, and unit-record files.
- Because GSAM is a sequential non-hierarchic database, it has no segments, keys, or parentage.

Java application programs running in JBP regions can access GSAM databases by using the IMS Java dependent region resource adapter.

Related concepts:

"Data areas in GSAM databases" on page 245

Related reference:

"Accessing GSAM data from a JBP application" on page 656

Accessing GSAM databases

The calls you use to access Generalized Sequential Access Method (GSAM) databases are different from those you use to access other IMS databases, and you can use GSAM databases for input and output.

For example, your program can read input from a GSAM database sequentially and then load another GSAM database with the output data. Programs that retrieve input from a GSAM database usually retrieve GSAM records sequentially and then process them. Applications that send output to a GSAM database must add output records to the end of the database as the program processes the records. You cannot delete or replace records in a GSAM database, and any records that you add must go at the end of the database.

PCB masks for GSAM databases

For the most part, you process GSAM databases in the same way that you process other IMS databases. You use calls that are very similar to DL/I calls to communicate your requests. GSAM describes the results of those calls in a GSAM DB PCB.

© Copyright IBM Corp. 1974, 2015

Calls to GSAM databases can use either the AIBTDLI or the PCB interface.

The DB PCB mask for a GSAM database serves the same purpose as it does for other IMS databases. The program references the fields of the DB PCB through the GSAM DB PCB mask. The GSAM DB PCB mask must contain the same fields as the GSAM DB PCB and must be of the same length.

Some differences exist between a DB PCB for a GSAM database and one for other IMS databases. Some of the fields are different, and the GSAM DB PCB has one field that the other PCBs do not. Because GSAM is not a hierarchical database, some fields in a PCB mask for other IMS databases do not have meanings in a GSAM PCB mask. The fields that are not used when you access GSAM databases are:

- The second field: segment level number
- The sixth field: segment name
- The eighth field: number of sensitive segments

Even though GSAM does not use these fields, you must define them in the order and length shown in the following table in the GSAM DB PCB mask.

When you code the fields in a DB PCB mask, name the area that contains all the fields as you do for a DB PCB. The entry statement associates each DB PCB mask in your program with a DB PCB in your program's PSB based on the order of the PCBs in the PSB. The entry statement refers to the DB PCB mask in your program by the name of the mask or by a pointer.

When you code the entry statement in:

- COBOL, Java, Pascal, C, and assembler language programs, the entry statement must list the names of the DB PCB masks in your program.
- PL/I programs, the entry statement must list the pointers to the DB PCB masks in your program.

The first PCB name or pointer in the entry statement corresponds to the first PCB. The second name or pointer in the entry statement corresponds to the second PCB, and so on.

Table 45. GSAM DB PCB mask

Descriptor	Byte length	DB/DC	DBCTL	DCCTL	DB batch	TM batch
Database name[1]	8	X	X	X	X	X
Segment level number[2]	2	N/A	N/A	N/A	N/A	N/A
Status code[3]	2	X	X	X	X	X
Processing options[4]	4	X	X	X	X	X
Reserved for IMS[5]	4	X	X	X	X	X
Segment name[6]	8	N/A	N/A	N/A	N/A	N/A
Length of key feedback area and undefined-length records area[7]	4	X	X	X	X	X
Number of sensitive segments[8]	4	N/A	N/A	N/A	N/A	N/A

Table 45. GSAM DB PCB mask (continued)

Descriptor	Byte length	DB/DC	DBCTL	DCCTL	DB batch	TM batch
Key feedback area[9]	8 or 12 for large data sets.	X	X	X	X	X
Length of undefined-length records[10]	4	X	X	X	X	X

Note:

1. **Database Name**. The name of the GSAM DBD. This field is 8 bytes and contains character data.
2. **Segment Level Number**. Not used by GSAM, but you must code it. It is 2 bytes.
3. **Status Code**. IMS places a two-character status code in this field after each call to a GSAM database. This code describes the results of the call. IMS updates this field after each call and does not clear it between calls. The application program should test this field after each call to find out whether the call was successful. If the call was completed successfully, this field contains blanks.
4. **Processing Options**. This is a 4-byte field containing a code that tells IMS the types of calls this program can issue. It is a security mechanism in that it can prevent a particular program from updating the database, even though the program can read the database. This value is coded in the PROCOPT parameter of the PCB statement when generating the PSB for the application program. The value does not change. For GSAM, the values are G, GS, L, or LS.
5. **Reserved for IMS**. This 4-byte field is used by IMS for internal linkage. It is not used by the application program.
6. **Segment Name**. This field is not used by GSAM, but it must be coded as part of the GSAM DB PCB mask. It is 8 bytes.
7. **Length of Key Feedback Area and Undefined-Length Records Area**. This is a 4-byte field that contains the decimal value of 12 (or 16 for large format data sets). This is the sum of the lengths of the Key Feedback Area and Undefined-Length Records Area.
8. **Number of Sensitive Segments**. This field is not used by GSAM, but it should be coded as part of the GSAM DB PCB mask. This field is 4 bytes.
9. **Key Feedback Area**. After a successful retrieval call, GSAM places the address of the record that is returned to your program in this field. This is called a record search argument (RSA). You can use it later if you want to retrieve that record directly by including it as one of the parameters on a `GU` call. This field is 8 bytes for basic format data sets or 12 bytes for large format data sets.
10. **Undefined-Length Records Area**. If you use undefined-length records (RECFM=U), the length in binary of the record you are processing is passed between your program and GSAM in this field. This field is 4 bytes long. When you issue a `GU` or `GN` call, GSAM places the binary length of the retrieved record in this field. When you issue an `ISRT` call, put the binary length of the record you are inserting in this field before issuing the `ISRT` call.

Related concepts:

"AIBTDLI interface" on page 245

"GSAM record formats" on page 310

Retrieving and inserting GSAM records

GSAM records can be retrieved sequentially or directly. You can also add GSAM records to a new data set or add new records to the end of an existing data set in the database.

To retrieve GSAM records sequentially, use the GN call. The only required parameters are the GSAM PCB and the I/O area for the segment. To process the whole database, issue the GN call until you get a GB status code in the GSAM PCB. This status code means that you have reached the end of the database. GSAM automatically closes the database when you reach the end of it. To add records to a new data set or to add new records to the end of an existing data set in the database, use the ISRT call. GSAM adds the records sequentially in the order in which you supply them.

You can retrieve records directly from a GSAM database by supplying a record search argument (RSA) to the GSAM database. An RSA is like a segment search argument (SSA), but it contains the exact address of the record that you want to retrieve. The specific contents and format of the RSA depend on the access method that GSAM is using. For BSAM tape data sets and VSAM data sets, the RSA contains the relative byte address (RBA). For BSAM disk data sets, the RSA contains the disk address and uses the relative track and record format.

The following table provides more details about the format of the RSA for basic format data sets:

Table 46. Format of the RSA for basic format data sets

Position	Address
Positions 1-4	• BSAM (DASD) relative track and record (TTRZ) for the block in the buffer. • BSAM RBA. • VSAM RBA.
Position 5	Relative data set of the concatenated data set. The first data set number is 1.
Position 6	Relative volume of the data set. The first volume of data set is 1.
Positions 7 and 8	The current displacement.

The following table provides more details about the format of the RSA for large format data sets:

Table 47. Format of the RSA for large format data sets

Position	Address
Positions 1-4	• BSAM (DASD) relative track and record (TTTR) for the block in the buffer. • BSAM RBA.
Position 5	Zone byte
Position 6	Relative data set of the concatenated data set. The first data set number is 1.

Table 47. Format of the RSA for large format data sets (continued)

Position	Address
Position 7	Relative volume of the data set. The first volume of data set is 1.
Positions 8-10	Null bytes. Not used.
Positions 11-12	The current displacement.

Before you can supply an RSA in a GU call to a GSAM database, that RSA must have previously been returned to you as a result of a GN or ISRT call. For GSAM to return an RSA, the GN or ISRT call must be issued with a fourth parameter that points to an 8-byte RSA save area in your program. Save this RSA until you want to retrieve that particular record.

To retrieve that particular record, issue a GU call for the record and specify the address of its RSA as a fourth parameter of the GU call. GSAM returns the record to the I/O area that you named as one of the call parameters.

Restriction: Retrieve records directly from a GSAM database on DASD only. When using buffered I/O, buffer definitions for the output PCB may affect performance.

Resetting the position in a GSAM Database

You can use the GU call to reset the position in the GSAM database.

You can reset the position to the start of the GSAM database or to a specific segment in the GSAM database:

- To reset the position to the start of the GSAM database using basic format data sets, issue a GU call with an RSA that consists of a fullword with a binary value of 1, followed by a fullword with a binary value of 0.
- To reset the position to the start of the GSAM database using large format data sets, issue a GU call with an RSA that consists of a fullword with a binary value of 1, followed by two fullwords with a binary value of 0.
- To reset the position to a specific segment in the GSAM database, issue a GU call with an RSA that contains the saved RSA value from a prior ISRT or GN call for that segment.

Related reference:

"GSAM coding considerations" on page 312

INIT call (Application Programming APIs)

Explicit open and close calls to GSAM

IMS opens the GSAM data set when the first call is made and closes the data set when the application program terminates. Therefore, the application program does not usually need to make explicit open or close calls to GSAM.

However, explicit OPEN and CLSE calls are useful if:

- the application program loads a GSAM data set, and then in the same step reads the data set using GSAM (for example, to sort the data set). The application program should issue the GSAM CLSE call after the load is complete.
- the GSAM data set is an output data set, and it is possible that when the program executes it does not make GSAM ISRT calls. A data set is not created. Subsequent attempts to read the nonexistent data set (using GSAM or not) will

likely result in an error. To avoid this situation, explicitly open the data set. DL/I closes the data set when the step terminates. Closing the data set prevents the possibility of attempting to read an empty data set.

- the GSAM data set is an output data set, and data exists beyond the EOF address in the dataset control block (DSCB). The previous job/step may have ended abnormally before the DSCB could be updated. If the program is restarted, but does not make GSAM ISRT calls, the EOF will not be updated at job/step termination when DL/I closes the data set. This could strand any data that exists past the EOF address. To avoid this situation, explicitly open the data set so that the DSCB can be updated with the correct EOF address.

The explicit OPEN or CLSE call need not include an I/O area parameter. Depending on the processing option of the PCB, the data set is opened for input or output. You can specify that an output data set contain either ASA or machine control characters. Including an I/O area parameter in the call and specifying OUTA in the I/O area indicates ASA control characters. Specifying OUTM specifies machine control characters.

GSAM record formats

GSAM records are nonkeyed. For variable-length records you must include the record length as the first 2 bytes of the record. Undefined-length records, like fixed-length records, contain only data (and control characters, if needed).

If you use undefined-length records, record length is passed between your program and GSAM in the 4-byte field that follows the key feedback area of the GSAM DB PCB. It is called the undefined-length records area. When you issue an ISRT call, supply the length. When you issue a GN or GU call, GSAM places the length of the returned record in this field. The advantage of using undefined-length records is that you do not need to include the record length at the beginning of the record, and records do not need to be of fixed length. The length of any record must be less than or equal to the block size (BLKSIZE) and greater than 11 bytes (an z/OS convention).

If you are using VSAM, you can use blocked or unblocked fixed-length or variable-length records. If you are using BSAM, you can use blocked or unblocked fixed-length, variable-length, or undefined-length records. Whichever you use, be sure to specify this on the RECFM keyword in the DATASET statement of the GSAM DBD. You can override this in the RECFM statement of the DCB parameter in the JCL. You can also include carriage control characters in the JCL for all formats.

Related concepts:

"PCB masks for GSAM databases" on page 305

"Origin of GSAM data set characteristics" on page 313

GSAM I/O areas

If you provide an optional I/O area, it must contain one of these values.

- INP for an input data set
- OUT for an output data set
- OUTA for an output data set with ASA control characters
- OUTM for an output data set with machine control characters

For GN, ISRT, and GU calls, the format of the I/O area depends on whether the record is fixed-length, undefined-length (valid only for BSAM), or variable-length. For each kind of record, you have the option of using control characters.

The formats of an I/O area for fixed-length or undefined-length records are:

- With no control characters, the I/O area contains only data. The data begins in byte 0.
- With control characters, the control characters are in byte 0 and the data begins in byte 1.

If you are using undefined-length records, the record length is passed between your program and GSAM in the PCB field that follows the key feedback area. When you are issuing an ISRT call, supply the length. When you are issuing a GN or GU call, GSAM places the length of the returned record in this field. This length field is 4 bytes long.

The formats for variable-length records differ because variable-length records include a length field, which other records do not have. The length field is 2 bytes. Variable-length I/O areas, like fixed-length and undefined-length I/O areas, can have control characters.

- Without control characters, bytes 0 and 1 contain the 2-byte length field, and the data begins in byte 2.
- With control characters, bytes 0 and 1 still contain the length field, but byte 2 contains the control characters, and the data starts in byte 3.

GSAM status codes

Your program should test for status codes after each GSAM call, just as it does after each DL/I or system service call.

If, you find that you have an error and terminate your program after checking the status codes, be sure to note the PCB in error before you terminate. The GSAM PCB address is helpful in determining problems. When a program that uses GSAM terminates abnormally, GSAM issues PURGE and CLSE calls internally, which changes the PCB information.

Status codes that have specific meanings for GSAM are:

AF GSAM detected a BSAM variable-length record with an invalid format. Terminate your program.

AH You have not supplied an RSA for a GU call.

AI There has been a data management OPEN error.

AJ One of the parameters on the RSA that you supplied is invalid.

AM You have issued an invalid request against a GSAM database.

AO An I/O error occurred when the data set was accessed or closed.

GB You reached the end of the database, and GSAM has closed the database. The next position is the beginning of the database.

IX You issued an ISRT call after receiving an AI or AO status code. Terminate your program.

Symbolic CHKP and XRST with GSAM

To checkpoint GSAM databases, use symbolic CHKP and XRST calls.

By using GSAM to read or write the data set, symbolic CHKP and XRST calls can be used to reposition the data set at the time of restart, enabling you to make your program restartable. When you use an XRST call, IMS repositions GSAM databases for processing. CHKP and XRST calls are available to application programs that can run as batch programs, batch-oriented BMPs, or transaction-oriented BMPs.

Restriction: When restarting GSAM databases:

- You cannot use temporary data sets with a symbolic CHKP or XRST call.
- A SYSOUT data set at restart time may give duplicate output data.
- You cannot restart a program that is loading a GSAM or VSAM database.
- The GSAM database data set must have the same data set format (BASIC or LARGE) as when the symbolic CHKP call was issued.

When IMS restores the data areas specified in the XRST call, it also repositions any GSAM databases that your program was using when it issued the symbolic CHKP call. If your program was loading GSAM databases when the symbolic CHKP call was issued, IMS repositions them (if they are accessed by BSAM). If you make a copy of the GSAM data set for use as input to the restart process, ensure that the short blocks are written to the new data set as short blocks, for example, using IEBGENER with RECFM=U for SYSUT1. You can also do the restart using the original GSAM data set.

During GSAM XRST processing, a check is made to determine if the GSAM output data set to be repositioned is empty, and if the abending job had previously inserted records into the data set.

GSAM coding considerations

The calls your program uses to access GSAM databases are not the same as the DL/I calls. The system service calls that you use with GSAM are symbolic CHKP and XRST.

The following table summarizes GSAM database calls. The five calls you can use to process GSAM databases are:

- CLSE
- GN
- GU
- ISRT
- OPEN

The COBOL, PL/I, Pascal, C, and assembler language call formats and parameters for these calls are the same and are described in the following table. GSAM calls do not differ significantly from DL/I calls, but GSAM calls must reference the GSAM PCB, and they do not use SSAs.

Java application programs running in Java batch processing (JBP) regions can access GSAM databases by using the IMS Java dependent region resource adapter.

Table 48. Summary of GSAM calls

Call Formats	Meaning	Use	Options	Parameters
CLSE	Close	Explicitly closes GSAM database	None	function, gsam pcb
GN△△	Get Next	Retrieves next sequential record	Can supply address for RSA to be returned	function, gsam pcb, i/o area [,rsa name]
GU△△	Get Unique	Establishes position in database or retrieves a unique record	None	function, gsam pcb, i/o area, rsa name
ISRT	Insert	Adds new record at end of database	Can supply address for RSA to be returned	function, gsam pcb, i/o area [,rsa name]
OPEN	Open	Explicitly opens GSAM database	Can specify printer or punch control characters	function, gsam pcb [, open option]

Related reference:

"Accessing GSAM data from a JBP application" on page 656

Origin of GSAM data set characteristics

For an input data set, the record format (RECFM), logical record length (LRECL), and block size (BLKSIZE) are based on the input data set label.

If this information is not provided by a data set label, the DD statement or the DBD specifications are used. The DD statement has priority.

An output data set can have the following characteristics:

- Record format
- Logical record length
- Block size
- Other JCL DCB parameters
- DNS type

Specify the record format on the DATASET statement of the GSAM DBD. The options are:

- V for variable
- VB for variable blocked
- F for fixed
- FB for fixed blocked
- U for undefined

The V, F, or U definition applies and is not overridden by the DCB=RECFM= specification on the DD statement. However, if the DD RECFM indicates blocked and the DBD does not, RECFM is set to blocked. If the DD RECFM of A or M control character is specified, it applies as well.

Unless an undefined record format is used, specify the logical record using the RECORD= parameter of the DATASET statement of DBDGEN, or use

DCB=LRECL=xxx on the DD statement. If the logical record is specified on both, the DD statement has priority. Refer to the following table for the maximum record length

Table 49. BSAM and VSAM logical record lengths for GSAM data sets by record format

Record Format	BSAM logical record length	VSAM logical record length
Fixed/Fixed Block	32760 bytes	32760 bytes
Variable/Variable Blocked	32756 bytes	32756 bytes
Undefined	32760 bytes	not supported

Specify block size using the BLOCK= or SIZE= parameter of the DATASET statement of DBDGEN, or use DCB=BLKSIZE=xxx on the DD statement. If block size is specified on both, the DD statement has priority. If the block size is not specified by the DBD or the DD statement, the system determines the size based on the device type, unless the undefined record format is used.

The other JCL DCB parameters that can be used, include:

- CODE
- DEN
- DNSTYPE
- TRTCH
- MODE
- STACK
- PRTSP, which can be used if RECFM does not include A or M
- DCB=BUFNO=X, which, when used, causes GSAM to use X number of buffers

Restriction: Do not use BFALN, BUFL, BUFOFF, FUNC, NCP, and KEYLEN.

Related concepts:

"GSAM record formats" on page 310

DD statement DISP parameter for GSAM data sets

The DD statement DISP parameter varies, depending on whether you are creating input or output data sets and how you plan to use the data sets.

Attention: Specifying the DISP=OLD or DISP=SHR parameter for a normal start with non-empty data sets will overwrite the existing records from the beginning of the data set.

- For input data sets, use the DISP=OLD parameter.
- For output data sets, consider the following options:
  - To create an output data set allocated by the DD statement, set DISP=NEW.
  - To add new records to an empty data set when performing normal start or a restart after failure, set DISP=MOD, DISP=SHR, or DISP=OLD.
  - When restarting the step, set DISP=OLD for existing data sets and DISP=MOD for empty data sets.
  - To add new records to an existing non-empty data set when performing a restart after failure, set DISP=MOD, DISP=SHR, or DISP=OLD. These parameters add new records from the restart point on the existing data set.
  - To add new records to the end of an existing non-empty data set when performing normal start, set DISP=MOD.

Extended checkpoint restart for GSAM data sets

If you are using extended checkpoint restart for GSAM data sets, these recommendations may apply.

- Do not use passed data sets.
- Do not use backward references to data sets in previous steps.
- Do not use DISP=MOD to add records to an existing tape data set.
- Do not use DISP=DELETE or DISP=UNCATLG.
- Use DFSMS striped data sets under the following conditions:
  - When the data sets is managed by SMS.
  - When the data sets are likely to exceed the system extent limit for volumes.
- Additionally, keep in mind that:
  - No attempt is made to reposition a SYSIN, SYSOUT, or temporary data set.
  - No attempt is made to reposition any of the concatenated data sets for a concatenated DD statement if any of the data sets are a SYSIN or SYSOUT.
  - If you are using concatenated data sets, specify the same number and sequence of data sets at restart time and checkpoint time.
  - GSAM/VSAM load mode restrictions apply to both non-striped and striped data sets.
  - If the PSB contains an open GSAM VSAM output data set when the symbolic checkpoint call is issued, the system returns an AM status code in the database PCB as a warning. This means that the data set is not repositioned at restart and the checkpoint has completed normally.
  - If an ISRT call is issued after a CLSE call and the GSAM data set is defined as DISP=OLD, all CHKP calls made prior to the CLSE call will contain invalid reposition information. Ensure a CHKP call is issued after a CLSE all when using DISP=OLD to avoid an abend U0271 after an extended restart (XRST).

Copying GSAM data sets between checkpoint and restart

To position GSAM data sets when restarting non-striped GSAM DASD data sets, use the relative track and record format (TTRZ or TTTRZ for large format data sets).

GSAM uses the TTRZ or TTTRZ on the volume to position non-striped GSAM DASD data sets when restarting. For a tape data set, the relative record on the volume is used. The relative record on the tape volume cannot be changed.

To copy non-striped DASD data sets between checkpoint and restart:

- Copy the data set to the same device type.
- Avoid any reblocking by using the undefined record format (RECFM=U) for both the input and the output data set.

Each copied volume contains the same number of records as the original volumes.

Note: GSAM uses the relative block number (RBN) to reposition striped DASD data sets. When data sets that are managed by SMS are used with GSAM databases, you cannot control how each volume is copied. After the data set is copied, unlike with non-striped DASD data sets, you do not need to ensure that the TTRZ or the TTTRZ of the restart record is unchanged.

Converting data sets from non-striped data sets to striped data sets

Convert GSAM/BSAM non-striped data sets to striped data sets before you must perform an extended restart when a system allocation limit is exceeded or a system X'37' error condition occurs. Non-striped data sets that are not managed by SMS extend beyond their initial primary or secondary allocation only by volume, but with non-striped GSAM/BSAM multiple volume data sets that are managed by SMS, the resulting new space allocation takes effect for all of the volumes in the data set.

If you copy non-striped data sets that are managed by SMS after you change the space allocation values, the number of records in the new volumes will be different from the number of records in the old volume. The new primary and secondary allocation values are used with non-striped data sets. As the data is copied, all of the space that is allocated on the new volume is used before the data is copied to the next volume.

If an error condition (system X'37' or system allocation limit exceeded) occurs during the processing of a GSAM/BSAM non-striped data set, and the data set is converted to a striped data set after the error occurs, a restart after failure will not complete successfully. Because the issued checkpoint saved a TTRZ or a TTTRZ value in the log record for repositioning, the log record for striped data sets will be used by GSAM restart after failure, which requires a relative block number (RBN) to perform the repositioning.

Concatenated data sets used by GSAM

GSAM can use concatenated data sets, which may be on unlike device types, such as DASD and tape, or on different DASD devices. Logical record lengths and block sizes can differ, and it is not required that the data set with the largest block size be concatenated first.

The maximum number of concatenated data sets for a single DD statement is 255. The number of buffers determined for the first of the concatenated data sets is used for all succeeding data sets. Generation data groups can result in concatenated data sets.

Specifying GSAM data set attributes

When specifying GSAM data set attributes, the following settings are recommended.

- On the DBD, specify RECFM. (It is required.)
- On the DATASET statement, specify the logical record length using RECORD=. If the data set can become larger than 65535 tracks on a DASD volume and you want the data set to not span multiple volumes, specify the DSNTYPE=LARGE parameter.
- On the DD statement, do not specify LRECL, RECFM, or BLKSIZE. The system determines block size, with the exception of RECFM=U. The system determines logical record length from the DBD.
- For the PSB, specify PROCOPT=LS for output and GS for input. If you include S, GSAM uses multiple buffers instead of a single buffer for improved performance.

IMS will add 2 bytes to the record length value specified in the DBD in order to accommodate the ZZ field that is needed to make up the BSAM RDW. Whenever

the database is GSAM or BSAM and the records are variable (V or VB), IMS will add 2 bytes to the record length value in the GSAM records passed by the application. Such addition allows IMS to accommodate the ZZ field that makes up the BSAM RDW (Record Descriptor Word).

Example of GSAM or BSAM where the records are variable

```
 //IDASD DD DUMMY
//ODASD DD UNIT=SYSDA,VOL=SER=000000,DISP=(,KEEP),
// SPACE=(TRK,(5,1)),DSN=GSAM.VARIABLE1,
// DCB=(RECFM=VB,BLKSIZE=32760,LRECL=32756)
//SYSIN DD *,DCB=BLKSIZE=80
S 1 1 1 1 1 DBDNAME
L ISRT
L V8187 DATA 1ST RECORD LOADED TO GSAM <---RDW
L ISRT
L V8187 DATA 2ND RECORD LOADED TO GSAM
L ISRT
L V8187 DATA 3RD RECORD LOADED TO GSAM
L ISRT
L V8187 DATA 4TH RECORD LOADED TO GSAM
```

In the above example, four GSAM records (IMS segment) can be contained in one 32756 byte (MVS) record.

DLI, DBB, and BMP region types and GSAM

To access GSAM databases, IMS builds its DLI control blocks using PSB and DBD information from PSBLIB, DBDLIB and ACBLIB. The source of the PSB and DBD information depends on the region type.

For DLI offline batch regions, IMS obtains PSB and DBD information from PSBLIB and DBDLIB. For DBB offline batch regions, IMS database management obtains PSB and DBD information from ACBLIB. For online batch regions (BMPs), IMS builds its DLI control blocks with information from ACBLIB. If an application is scheduled in a BMP region and the PSB associated with the application contains one or more GSAM PCBs, IMS scheduling obtains PSB information from ACBLIB and PSBLIB. In this case, the PSB in ACBLIB and PSBLIB must be the same. GSAM database management does not obtain PSB and DBD information from ACBLIB. Instead, GSAM database management obtains PSB and DBD information from PSBLIB and DBDLIB.

When you initialize a DLI, DBB or BMP region using GSAM, you must include an //IMS DD and GSAM DD statements. When DBB or BMP regions are not using GSAM, //IMS DD statements do not need to be included. To load PSBs and DBDs and build GSAM control blocks, you must include an //IMS DD statement. In the following figure, an example of the //IMS DD statement with data sets that are larger than 65535 tracks is shown.

Figure 64. //IMS DD statement example

```
//STEP       EXEC     PGM=DFSRRC00,PARM=[BMP|DBB|DLI],...'
//STEPLIB    DD       DSN=executionlibrary-name,DISP=SHR
//           DD       DSN=pgmlib-name,DISP=SHR
//IMS        DD       DSN=psblib-name,DISP=SHR
//           DD       DSN=dbdlib-name,DISP=SHR
//IMSACB     DD       DSN=acblib-name,disp=shr (required for DBB)
//SYSPRINT   DD       SYSOUT=A
//SYSUDUMP   DD       SYSOUT=A
//ddnamex    DD       (add DD statements for required GSAM databases)
//ddnamex    DD       (add DD statements for non-GSAM IMS databases
```

```
                        for DLI/DBB)
//ddnamex    DD         DSNTYPE=LARGE,...
             .
             .
             .
/*
```

Chapter 20. Processing Fast Path databases

You can write application programs to access Fast Path databases, including main storage databases and data entry databases.

The two kinds of Fast Path databases are:

- Main storage databases (MSDBs), which are available in a DB/DC environment, and contain only root segments in which you store data that you access most frequently.
- Data entry databases (DEDBs) are hierarchic databases that can have as many as 15 hierarchic levels and as many as 127 segment types. DEDBs are available to both IMS users and CICS users with DBCTL.

Restriction: This DEDB information applies to CICS users with DBCTL. CICS users can access MSDBs in DBCTL in read mode, but update mode is not supported.

VSO considerations

VSO is transparent to the processing of an application. Where the data resides is immaterial to the application.

Data locking for MSDBs and DEDBs

All MSDB calls, including the FLD call, can lock the data at the segment level. The lock is acquired at the time the call is processed and is released at the end of the call. All DEDB calls, with the exception of HSSP calls, are locked at the VSAM CI level. For single-segment, root-only, fixed-length VSO areas, if you specify PROCOPT R or G, the application program can obtain segment-level locks for all calls. If you specify any other PROCOPT, the application program obtains VSAM CI locks.

Segment-level locking (SLL) provides a two-tier locking scheme. First, a share (SHR) lock is obtained for the entire CI. Then, an exclusive (EXCL) segment lock is obtained for the requested segment. This scheme allows for contention detection between SLL users of the CI and EXCL requestors of the CI. When contention occurs between an existing EXCL CI lock user and a SHR CI lock requestor, the SHR CI lock is upgraded to an EXCL CI lock. During the time that this EXCL CI lock is held, subsequent SHR CI lock requests must wait until the EXCL CI is released at the next commit point.

DEDB FLD calls are not locked at call time. Instead, the lock is acquired at a commit point.

During sync-point processing, the lock is re-acquired (if not already held), and the changes are verified. Verification failure results in the message being reprocessed (for message-driven applications) or an FE status code (for non-message-driven applications). Verification can fail if the segment used by the FLD call has been deleted or replaced before a sync-point.

Segment retrieval for a FLD call is the same as for a GU call. An unqualified FLD call returns the first segment in the current area, just as an unqualified GU call does.

© Copyright IBM Corp. 1974, 2015

After the FLD call is processed, all locks for the current CI are released if the current CI is unmodified by any previous call.

When a compression routine is defined on the root segment of a DEDB with a root-only structure, and when that root segment is a fixed-length segment, its length becomes variable after being compressed. To replace a compressed segment, you must perform a delete and an insert. In this case, segment level control and locking will not be available.

Related concepts:

Data entry databases (Database Administration)

Main storage databases (MSDBs) (Database Administration)

High-speed sequential processing (HSSP) (Database Administration)

Fast Path database calls

Use Fast Path database calls in your application programs to access Fast Path databases.

The following table summarizes the database calls you can use with Fast Path databases.

Table 50. Summary of Fast Path database calls.

	Types of MSDBs:			
Function Code	**Nonterminal-Related**	**Terminal-Related Fixed**	**Terminal-Related Dynamic**	**DEDBs**
DEQ				X
FLD	X	X	X	X
GU, GHU	X	X	X	X
GN, GHN	X	X	X	X
GNP, GHNP DLET			X	X
ISRT			X	X
POS				X
REPL	X	X	X	X
RLSE				X

DL/I calls to DEDBs can include the same number of SSAs as existing levels in the hierarchy (a maximum of 15). They can also include command codes and multiple qualification statements.

Restriction:

- Fast Path ignores command codes that are used with sequential dependent segments.
- If you use a command code that does not apply to the call you are using, Fast Path ignores the command code.
- If you use F or L in an SSA for a level greater than the established parent, Fast Path ignores the F or L command code.

- DL/I calls to DEDBs cannot include the independent AND, which is used only with secondary indexing.

Calls to DEDBs can use all command codes. Only calls to DEDBs that use subset pointers can use the R, S, Z, W, and M command codes. The following table shows which calls you can use with these command codes.

Table 51. Subset pointer command codes and calls

Command Code	DLET	GU GHU	GN GHN	GNP GHNP	ISRT	REPL
M		X	X	X	X	X
R		X	X	X	X	
S		X	X	X	X	X
W		X	X	X	X	X
X	X	X	X	X	X	X

Main storage databases (MSDBs)

MSDBs contain only root segments. Each segment is like a database record, because the segment contains all of the information about a particular subject.

In a DL/I hierarchy, a database record is made up of a root segment and all its dependents. For example, in the medical hierarchy, a particular PATIENT segment and all the segments underneath that PATIENT segment comprise the database record for that patient. In an MSDB, the segment is the whole database record. The database record contains only the fields that the segment contains. MSDB segments are fixed length.

The two kinds of MSDBs are terminal related and non-terminal related. In terminal-related MSDBs, each segment is owned by one logical terminal. The segment that is owned can be updated only by that terminal. Related MSDBs can be fixed or dynamic. You can add segments to and delete segments from dynamic related MSDBs. You cannot add segments to or delete segments from fixed related MSDBs.

In the second kind of MSDB, called non-terminal related (or nonrelated) MSDBs, the segments are not owned by logical terminals.

Restrictions on using calls for MSDBs

To retrieve segments from an MSDB, you can issue Get calls just as you do to retrieve segments from other IMS databases. Because MSDBs contain only root segments, you only use `GU` and `GN` calls (and `GHU` and `GHN` calls when you plan to update a segment). If the segment name field in the SSA contains *MYLTERM, the `GU`, `GHU`, and `FLD` calls return the LTERM-owned segment, and the remainder of the SSA is ignored.

When you are processing MSDBs, you should consider the following differences between calls to MSDBs and to other IMS databases:

- You can use only one SSA in a call to an MSDB.
- MSDB calls cannot use command codes.
- MSDB calls cannot use multiple qualification statements (Boolean operators).

- The maximum length for an MSDB segment key is 240 bytes (not 255 bytes, as in other IMS databases).
- If the SSA names an arithmetic field (types P, H, or F) as specified in the database description (DBD), the database search is performed using arithmetic comparisons (rather than the logical comparisons that are used for DL/I calls).
- If a hexadecimal field is specified, each byte in the database field is represented in the SSA by its two-character hexadecimal representation. This representation makes the search argument twice as long as the database field.

  Characters in hexadecimal-type SSA qualification statements are tested for validity before translation to the database format. Only numerals 0 through 9 and letters A through F are accepted.
- Terminal-related and non-terminal-related LTERM-keyed MSDBs are not supported for ETO or LU 6.2 terminals. Attempted access results in no data being retrieved and an AM status code. See *IMS Version 11 Communications and Connections* for more information on ETO and LU 6.2.
- MSDBs cannot be shared among IMS subsystems in a sysplex group. When using the Fast Path Expedited Message Handler (EMH), terminal related and non-terminal related with terminal key MSDBs can only be accessed by static terminals. These static terminals run transactions with Sysplex Processing Code (SPC) of Locals Only as specified in DBFHAGU0 (Input Edit Router exit routine).

The restrictions above do not apply to CICS users.

Data entry databases (DEDBs)

A DEDB contains a root segment and as many as 127 dependent segment types. One of these can be a sequential dependent; the other 126 are direct dependents. Sequential dependent segments are stored in chronological order. Direct dependent segments are stored hierarchically.

DEDBs can provide high data availability. Each DEDB can be partitioned, or divided into multiple areas. Each area contains a different collection of database records. In addition, you can make as many as seven copies of each area data set. If an error exists in one copy of an area, application programs continue to access the data by using another copy of that area. Use of the copy of an area is transparent to the application program. When an error occurs to data in a DEDB, IMS does not stop the database. IMS makes the data in error unavailable but continues to schedule and process application programs. Programs that do not need the data in error are unaffected.

DEDBs can be shared among application programs in separate IMS systems. Sharing DEDBs is virtually the same as sharing full-function databases, and most of the same rules apply. IMS systems can share DEDBs at the area level (instead of at the database level as with full-function databases), or at the block level.

Related reading: For more information on DEDB data sharing, see the explanation of administering IMS systems that share data in *IMS Version 11 System Administration*.

Updating segments: REPL, DLET, ISRT, and FLD

Three of the calls that you can use to update an MSDB or DEDB are the same ones that you use to update other IMS databases: `REPL`, `DLET`, and `ISRT`.

You can issue a REPL call to a related MSDB or nonrelated MSDB, and you can issue any of the three calls for non-terminal-related MSDBs (without terminal-related keys) or DEDBs. When you issue REPL or DLET calls against an MSDB or DEDB, you must first issue a Get Hold call for the segment you want to update, just as you do when you replace or delete segments in other IMS databases.

One call that you can use against MSDBs and DEDBs that you cannot use against other types of IMS databases is the Field (FLD) call, which enables you to access and change the contents of a field within a segment. The FLD call has two types:

- FLD/VERIFY

  This type of call compares the value of the field in the target segment to the value you supply in the FSA.
- FLD/CHANGE

  This type of call changes the value of the field in the target segment in the way that you specify in the FSA. A FLD/CHANGE call is only successful if the previous FLD/VERIFY call is successful.

The FLD call does in one call what a Get Hold call and a REPL call do in two calls. For example, using the ACCOUNT segment shown in the topic "Account Segment in a Nonrelated MSDB", a bank would need to perform the following processing to find out whether a customer could withdraw a certain amount of money from a bank account:

1. Retrieve the segment for the customer's account.
2. Verify that the balance in the account is more than the amount that the customer wants to withdraw.
3. Update the balance to reflect the withdrawal if the amount of the balance is more than the amount of the withdrawal.

Without using the FLD call, a program would issue a GU call to retrieve the segment, then verify its contents with program logic, and finally issue a REPL call to update the balance to reflect the withdrawal. If you use the FLD call with a root SSA, you can retrieve the desired segment. The FLD call has the same format as SSAs for other calls. If no SSA exists, the first segment in the MSDB or DEDB is retrieved. You use the FLD/VERIFY to compare the BALANCE field to the amount of the withdrawal. A FLD/CHANGE call can update the BALANCE field if the comparison is satisfactory.

The segment retrieved by a FLD call is the same as can be retrieved by a GHU call. After the FLD call, the position is lost. An unqualified GN call after a FLD call returns the next segment in the current area.

Checking the contents of a field: FLD/VERIFY

A FLD/VERIFY call compares the contents of a specified field in a segment to the value that you supply. The way that a FLD/VERIFY call compares the two depends on the operator you supply.

When you supply the name of a field and a value for comparison, you can determine if the value in the field is:

- Equal to the value you have supplied
- Greater than the value you have supplied
- Greater than or equal to the value you have supplied
- Less than the value you have supplied

- Less than or equal to the value you have supplied
- Not equal to the value you have supplied

After IMS performs the comparison that you have asked for, it returns a status code (in addition to the status code in the PCB) to tell you the results of the comparison.

You specify the name of the field and the value that you want its value compared to in a field search argument, or FSA. The FSA is also where IMS returns the status code. You place the FSA in an I/O area before you issue a FLD call, and then you reference that I/O area in the call—just as you do for an SSA in a DL/I call. An FSA is similar to an SSA in that you use it to give information to IMS about the information you want to retrieve from the database. An FSA, however, contains more information than an SSA. The table below shows the structure and format of an FSA.

Table 52. FSA structure

FSA Component	Field Length
FLD NAME	8
SC	1
OP	1
FLD VALUE	Variable
CON	1

The five fields in an FSA are:

Field Name (FLD Name)
This is the name of the field that you want to update. The field must be defined in the DBD.

Status Code (SC)
This is where IMS returns the status code for this FSA. If IMS successfully processes the FSA, it returns a blank status code. If IMS fails to process the FSA, it returns a FE status code to the PCB to indicate a nonblank status code in the FSA and returns a nonblank FSA status code. The FSA status codes that IMS might return to you on a FLD/VERIFY call are:

B The length of the data supplied in the field value is invalid, or the segment length of the data in the database is too small to contain the field length specified in the DBD.

D The verify check is unsuccessful. In other words, the answer to your query is no.

E The field value contains invalid data. The data you supplied in this field is not the same type of data that is defined for this field in the DBD.

H The requested field is not found in the segment.

Operator (OP)
This tells IMS how you want the two values compared. For a FLD/VERIFY call, you can specify:

E Verify that the value in the field is equal to the value you have supplied in the FSA.

G Verify that the value in the field is greater than the value you have supplied in the FSA.

H Verify that the value in the field is greater than or equal to the value you have supplied in the FSA.

L Verify that the value in the field is less than the value you have supplied in the FSA.

M Verify that the value in the field is less than or equal to the value you have supplied in the FSA.

N Verify that the value in the field is not equal to the value you have supplied in the FSA.

Field Value (FLD Value)
This area contains the value that you want IMS to compare to the value in the segment field. The data that you supply in this area must be the same type of data in the field you have named in the first field of the FSA. The five types of data are: hexadecimal, packed decimal, alphanumeric (or a combination of data types), binary fullword, and binary halfword. The length of the data in this area must be the same as the length that is defined for this field in the DBD.

Exceptions:

- If you are processing hexadecimal data, the data in the FSA must be in hexadecimal. This means that the length of the data in the FSA is twice the length of the data in the field in the database. IMS checks the characters in hexadecimal fields for validity before that data is translated to database format. (Only 0 to 9 and A to F are valid characters.)
- For packed-decimal data, you do not need to supply the leading zeros in the field value. This means that the number of digits in the FSA might be less than the number of digits in the corresponding database field. The data that you supply in this field must be in a valid packed-decimal format and must end in a sign digit.

When IMS processes the FSA, it does logical comparisons for alphanumeric and hexadecimal fields; it does arithmetic comparisons for packed decimal and binary fields.

Connector (CON)
If this is the only or last FSA in this call, this area contains a blank. If another FSA follows this one, this area contains an asterisk (*). You can include several FSAs in one FLD call, if all the fields that the FSAs reference are in the same segment. If you get an error status code for a FLD call, check the status codes for each of the FSAs in the FLD call to determine where the error is.

When you have verified the contents of a field in the database, you can change the contents of that field in the same call. To do this, supply an FSA that specifies a change operation for that field.

Changing the contents of a field: FLD/CHANGE

To indicate to IMS that you want to change the contents of a particular field, use an FSA, just as you do in a FLD/VERIFY call

. The difference is in the operators that you can specify and the FSA status codes that IMS can return to you after the call. To use FLD/CHANGE:

- You specify the name of the field that you want to change in the first field of the FSA (Field Name).

- You specify an operator in the third field of the FSA (Operator), which indicates to IMS how you want to change that field.
- You specify the value that IMS must use to change the field in the last area of the FSA (Field Value).

By specifying different operators in a FLD/CHANGE call, you change the field in the database in these ways:

- Add the value supplied in the FSA to the value in the field.
- Subtract the value supplied in the FSA from the value in the field.
- Set the value in the database field to the value supplied in the FSA.

You code these operators in the FSA with these symbols:

- To add: +
- To subtract: -
- To set the field equal to the new value: =

You can add and subtract values only when the field in the database contains arithmetic (packed-decimal, binary-fullword, or binary-halfword) data.

The status codes you can receive in a FLD/CHANGE FSA are:

A Invalid operation; for example, you specified the + operator for a field that contains character data.

B Invalid data length. The data you supplied in the FSA is not the length that is defined for that field in the DBD.

C You attempted to change the key field in the segment. Changing the key field is not allowed.

E Invalid data in the FSA. The data that you supplied in the FSA is not the type of data that is defined for this field in the DBD.

F You tried to change an unowned segment. This status code applies only to related MSDBs.

G An arithmetic overflow occurred when you changed the data field.

H The requested field was not found in the segment.

Example of using FLD/VERIFY and FLD/CHANGE

Using the bank account segment from the "Bank Account Example" database, assume that a customer wants to withdraw $100 from a checking account. The checking account number is 24056772. To find out whether the customer can withdraw this amount, you must check the current balance. If the current balance is greater than $100, you want to subtract $100 from the balance, and add 1 to the transaction count in the segment.

You can do all of this processing by using one FLD call and three FSAs. The three FSAs are described:

1. Verify that the value in the BALANCE field is greater than or equal to $100. For this verification, you specify the BALANCE field, the H operator for greater than or equal to, and the amount. The amount is specified without a decimal point. Field names less than eight characters long must be padded with trailing blanks to equal eight characters. You also have to leave a blank between the field name and the operator for the FSA status code. This FSA looks like this:

```
BALANCEbb
H10000*
```

The last character in the FSA is an asterisk, because this FSA will be followed by other FSAs.

2. Subtract $100 from the value in the BALANCE field if the first FSA is successful. If the first FSA is unsuccessful, IMS does not continue processing. To subtract the amount of the withdrawal from the amount of the balance, you use this FSA:

```
BALANCEbb
-10000*
```

Again, the last character in the FSA is an asterisk, because this FSA is followed by a third FSA.

3. Add 1 to the transaction count for the account. To do this, use this FSA:

```
TRANCNTbb
001b
```

In this FSA, the last character is a blank (b), because this is the last FSA for this call.

When you issue the FLD call, you do not reference each FSA individually; you reference the I/O area that contains all of them.

Commit-point processing in MSDBs and DEDBs

Your existing application programs can use either the MSDB commit view or the default DEDB commit view.

MSDB commit view

When you update a segment in an MSDB, IMS does not apply your updates immediately. Updates do not go into effect until your program reaches a commit point.

As a result of the way updates are handled, you can receive different results if you issue the same call sequence against a full-function database or a DEDB and an MSDB. For example, if you issue GHU and REPL calls for a segment in an MSDB, and then issue another Get call for the same segment in the same commit interval, the segment that IMS returns to you is the "old" value, not the updated one. If, however, you issue the same call sequence for a segment in a full-function database or DEDB, the second Get call returns the updated segment.

When the program reaches a commit point, IMS also reprocesses the FLD VERIFY/CHANGE call. If the VERIFY test passes, the change is applied to the database. If the VERIFY test fails, the changes made since the previous commit point are undone, and the transaction is reprocessed.

DEDBs with MSDB commit view

To use the MSDB commit view for DEDBs, specify VIEW=MSDB on the PCB statement; if you do not specify VIEW=MSDB, the DEDB uses the default DEDB commit view. So no changes to any existing application programs are required in order to migrate your MSDBs to DEDBs.

Assume that you specify VIEW=MSDB in the PCB and an application program issues GHU and REPL calls to a DEDB followed by another GHU call for the segment in the

same commit interval. Then the application program receives the old value of the data and not the new value from the REPL call. If you do not specify VIEW=MSDB, your application program receives the new updated values of the data, just as you expect for a DEDB or other DL/I database.

You can specify VIEW=MSDB for any DEDB PCB. If it is specified for a non-DEDB database, you receive message DFS0904 during ACBGEN.

If you issue a REPL call with a PCB that specifies VIEW=MSDB, the segment must have a key. This requirement applies to any segment in a path if command code 'D' is specified. Otherwise, the AM status code is returned. See *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes* for information about that status code.

The following code shows an example of a PCB that specifies the VIEW option.

Sample PCB specifying View=MSDB

```
PCB  ,                                                    *00000100
     TYPE=DB,                                             *00000200
     NAME=DEDBJN21,                                       *00000300
     PROCOPT=A,                                           *00000400
     KEYLEN=30,                                           *00000500
     VIEW=MSDB,                                           *00000600
     POS=M                                                 00000700
```

Related reference:

"Issuing checkpoints" on page 283

"Commit-point processing in a DEDB" on page 336

Processing DEDBs (IMS and CICS with DBCTL)

You can use subset pointers, the POS call, data locking, and the P and H processing options in your application program to process DEDBs.

Processing Fast Path DEDBs with subset pointer command codes

Subset pointers and the command codes you use with them are optimization tools that significantly improve the efficiency of your program when you need to process long segment chains.

Subset pointers are a means of dividing a chain of segment occurrences under the same parent into two or more groups or subsets. You can define as many as eight subset pointers for any segment type. You then define the subset pointers from within an application program. Each subset pointer points to the start of a new subset. For example, in the following topic, suppose you define one subset pointer that divides the last three segment occurrences from the first four. Your program can then refer to that subset pointer through command codes and directly retrieve the last three segment occurrences.

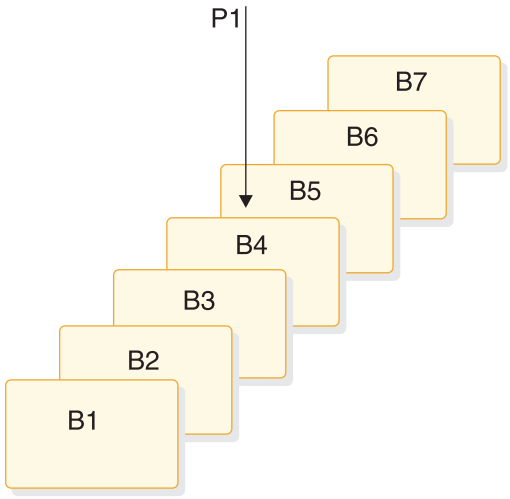


Figure 65. Processing a long chain of segment occurrences with subset pointers

You can use subset pointers at any level of the database hierarchy, except at the root level. If you try to use subset pointers at the root level, they are ignored.

The following figures show some of the ways you can set subset pointers. Subset pointers are independent of one another, which means that you can set one or more pointers to any segment in the chain. For example, you can set more than one subset pointer to a segment, as shown in the following figure.

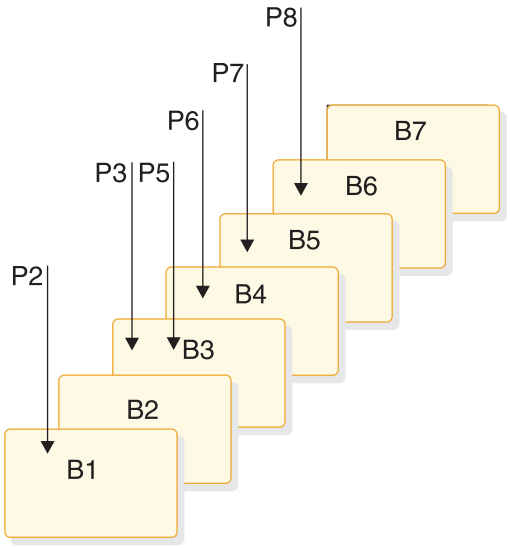


Figure 66. Examples of setting subset pointers

You can also define a one-to-one relationship between the pointers and the segments, as shown in the following figure.

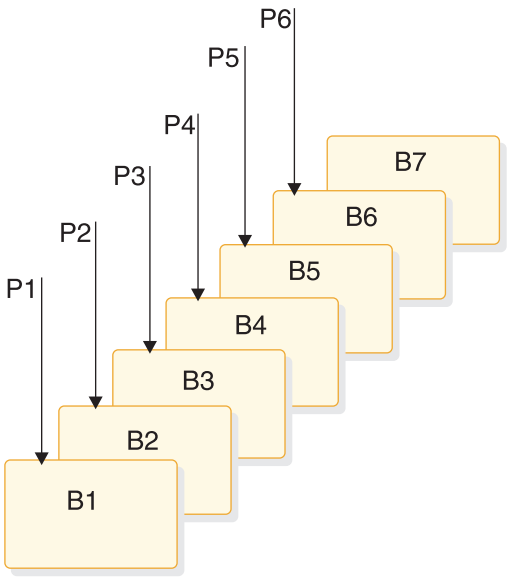


Figure 67. Additional examples of setting subset pointers

The following figure shows how the use of subset pointers divides a chain of segment occurrences under the same parent into subsets. Each subset ends with the last segment in the entire chain. For example, the last segment in the subset that is defined by subset pointer 1 is B7.

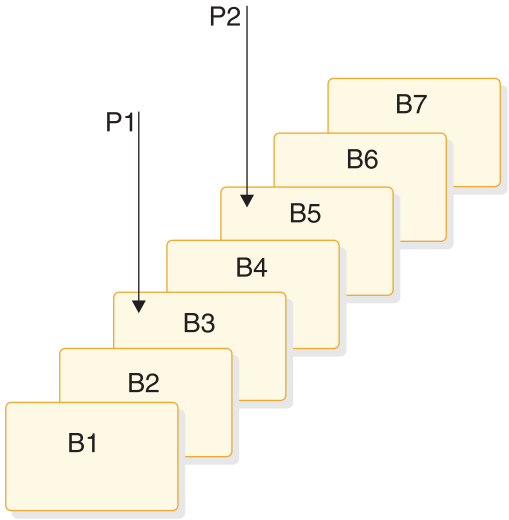


Figure 68. How subset pointers divide a chain into subsets

Before you use subset pointers

For your program to use subset pointers, the pointers must be defined in the DBD for the DEDB and in your program's PSB:

- In the DBD, you specify the number of pointers for a segment chain. You can specify as many as eight pointers for any segment chain.
- In the PSB, you specify which pointers your program is to use. Define this on the SENSEG statement. (Each pointer is defined as an integer from 1 to 8.) Also, indicate on the SENSEG statement whether your program can set the pointers it uses. If your program has read sensitivity, it cannot set pointers but can only

retrieve segments using subset pointers that are already set. If your program has update sensitivity, it can also update subset pointers by using the S, W, M, and Z command codes.

After the pointers are defined in the DBD and the PSB, an application program can set the pointers to segments in a chain. When an application program finishes executing, the subset pointers used by that program remain as they were set by the program; they are not reset.

Designating subset pointers

To use subset pointers in your program, you must know the numbers for the pointers as they were defined in the PSB. When you use the subset pointer command codes, specify the number of each subset pointer you want to use followed by the command code. For example, you use R3 to indicate that you want to retrieve the first segment occurrence in the subset defined by subset pointer 3. No default exists, so if you do not include a number between 1 and 8, IMS considers your SSA invalid and returns an AJ status code.

Subset pointer command codes

To take advantage of subsets, application programs use five command codes. The R command code retrieves the first segment in a subset. The following 4 command codes, which are explained in the topic "DEDB command codes for DL/I" in *IMS Version 11 Application Programming APIs*, redefine subsets by modifying the subset pointers:

Z Sets a subset pointer to 0.

M Sets a subset pointer to the segment following the current segment.

S Unconditionally sets a subset pointer to the current segment.

W Conditionally sets a subset pointer to the current segment.

Before your program can set a subset pointer, it must establish a position in the database. A call must be fully satisfied before a subset pointer is set. The segment a pointer is set to depends on your current position at the completion of the call. If a call to retrieve a segment is not completely satisfied and a position is not established, the subset pointers remain as they were before the call was made. You can use subset pointer command codes in either an unqualified SSA or a qualified SSA. To use a command code in a call with an unqualified SSA, use the command code along with the number of the subset pointer you want, after the segment name. This is shown in the following figure.

Table 53. Unqualified SSA with subset pointer command code

Seg Name	*	Cmd Code	Ssptr.	⌂
8	1	Variable	Variable	1

To use a subset pointer command code with a qualified SSA, use the command code and subset pointer number immediately before the left parenthesis of the qualification statement, as shown in the following figure.

Table 54. Qualified SSA with subset pointer command code

Seg Name	*	Cmd Code	Ssptr.	(	Fld Name	R.O.	Fld Value	)
8	1	Variable	Variable	1	8	2	Variable	1

Inserting segments in a subset

When you use the R command code to insert an unkeyed segment in a subset, the new segment is inserted before the first segment occurrence in the subset. However, the subset pointer is not automatically set to the new segment occurrence.

For example, the following call inserts a new B segment occurrence in front of segment B5, but does not set subset pointer 1 to point to the new B segment occurrence:

```
ISRT   A␣␣␣␣␣␣␣
Akey␣␣␣␣
=␣
A1)
       B␣␣␣␣␣␣␣
*R1
```

To set subset pointer 1 to the new segment, you use the S command code along with the R command code, as shown in the following example:

```
ISRT   A␣␣␣␣␣␣␣
(Akey␣␣␣␣
=␣A1)
       B␣␣␣␣␣␣␣
*R1S1
```

If the subset does not exist (subset pointer 1 is set to 0), the segment is added to the end of the segment chain.

Deleting the segment pointed to by a subset pointer

If you delete the segment pointed to by a subset pointer, the subset pointer points to the next segment occurrence in the chain. If the segment you delete is the last segment in the chain, the subset pointer is set to 0.

Combining command codes

You can use the S, M, and W command codes with other command codes, and you can combine subset pointer command codes with each other, as long as they do not conflict. For example, you can use R and S together, but you cannot use S and Z together because their functions conflict. If you combine command codes that conflict, IMS returns an AJ status code to your program.

You can use one R command code for each SSA and one update command code (Z, M, S, or W) for each subset pointer.

Related concepts:

"SSAs and command codes" on page 189

"Calls with dependent segments for DEDBs" on page 337

Subset pointer status codes

If you make an error in an SSA that contains subset pointer command codes, IMS can return either of these status codes to your program.

AJ The SSA used an R, S, Z, W, or M command code for a segment that does not have subset pointers defined in the DBD.

The subset command codes included in the SSA are in conflict. For example, if one SSA contains an S command code and a Z command code

for the same subset pointer, IMS returns an AJ status code. S indicates that you want to set the pointer to current position; Z indicates that you want to set the pointer to 0. You cannot use these command codes in one SSA.

The SSA includes more than one R command code.

The pointer number following a subset pointer command code is invalid. You either did not include a number, or you included an invalid character. The number following the command code must be between 1 and 8.

AM The subset pointer referenced in the SSA is not specified in the program's PSB. For example, if your program's PSB specifies that your program can use subset pointers 1 and 4, and your SSA references subset pointer 5, IMS returns an AM status code.

Your program tried to use a command code that updates the pointer (S, W, or M), but the program's PSB did not specify pointer-update sensitivity.

Your program attempted to open a GSAM database without specifying an IOAREA.

Retrieving location with the POS call (for DEDB only)

Use the POS (Position) call to retrieve the location of a specific sequential dependent segment; retrieve the location of the last-inserted sequential dependent segment, its time stamp, and the IMS ID; or retrieve the time stamp of a sequential dependent or Logical Begin. You can also use the POS call to tell the amount of unused space within each DEDB area. For example, you can use the information that IMS returns for a POS call to scan or delete the sequential dependent segments for a particular time period.

The topic "POS Call" in *IMS Version 11 Application Programming APIs* explains how you code the POS call and what the I/O area for the POS call looks like. If the area that the POS call specifies is unavailable, the I/O area is unchanged, and the FH status code is returned.

Locating a specific sequential dependent

When you have position on a particular root segment, you can retrieve the position information and the area name of a specific sequential dependent of that root. If you have a position established on a sequential dependent segment, the search starts from that position. IMS returns the position information for the first sequential dependent segment that satisfies the call. To retrieve this information, issue a POS call with a qualified or unqualified SSA containing the segment name of the sequential dependent. Current position after this kind of POS call is the same place that it would be after a GNP call.

After a successful POS call, the I/O area contains:

LL A 2-byte field giving the total length of the data in the I/O area, in binary.

Area Name
An 8-byte field giving the ddname from the AREA statement.

Position
An 8-byte field containing the position information for the requested segment.

Exception: If the sequential dependent segment that is the target of the POS call is inserted in the same synchronization interval, no position information is returned. Bytes 11-18 contain X'FF'. Other fields contain normal data.

Unused CIs
A 4-byte field containing the number of unused CIs in the sequential dependent part.

Unused CIs
A 4-byte field containing the number of unused CIs in the independent overflow part.

Locating the last inserted sequential dependent segment

You can also retrieve the position information for the most recently inserted sequential dependent segment of a given root segment. To do this, you issue a POS call with an unqualified or qualified SSA containing the root segment as the segment name. Current position after this type of call follows the same rules as position after a GU call.

You can also retrieve the position of the SDEP, its time stamp, and the ID of the IMS that owns the segment. To do this, you issue a POS call with a qualified SSA and provide the keyword PCSEGTSP in position one of the I/O area as input to the POS call. The keyword requests the POS call to return the position of the SDEP, its time stamp, and the ID of the IMS that owns the segment.

Requirement: The I/O area must be increased in size to 42 bytes to allow for the added data being returned. The I/O area includes a 2-byte LL field that is not shown in the following table. This LL field is described after the following table.

Table 55. Qualified POS call: keywords and map of I/O area returned

Keyword	word 0	word 1	word 2	word 3	word 4	word 5	word 6	word 7	word 8	word 9
<null>	Field 1		Field 2		Field 3	Field 4	N/A		N/A	
PCSEGTSP	Field 1		Field 2		Field 5		Field 6		Field 7	

Field 1 Area name

Field 2 Sequential dependent location from qualified SSA

Field 3 Unused CIs in sequential dependent part

Field 4 Unused CIs in independent overflow part

Field 5 Committed sequential dependent segment time stamp

Field 6 IMS ID

Field 7 Pad

After a successful POS call, the I/O area contains:

LL (Not shown in table) A 2-byte field, in binary, containing the total length of the data in the I/O area.

(Field 1)

Area Name
An 8-byte field giving the ddname from the AREA statement.

(Field 2)

Position
An 8-byte field containing the position information for the most recently inserted sequential dependent segment. This field contains zeros if no sequential dependent exists for this root.

Sequential dependent location from qualified SSA
IMS places two pieces of data in this 8-byte field after a successful POS call. The first 4 bytes contain the cycle count, and the second 4 bytes contain the VSAM RBA.

If the sequential dependent segment that is the target of the POS call is inserted in the same synchronization interval, no position information is returned. Bytes 11-18 contain X'FF'. Other fields contain normal data.

(Field 3)

Unused CIs in sequential dependent part
A 4-byte field containing the number of unused control intervals in the sequential dependent part.

(Field 4)

Unused CIs in independent overflow part
A 4-byte field containing the number of unused control intervals in the independent overflow part.

(Field 5)

Committed Sequential Dependent Segment Time Stamp
An 8-byte field containing the time stamp that corresponds to the SDEP segment located by the qualified POS call.

(Field 6)

IMS ID
Identifies the IMS that owns the CI where the SDEP segment was located.

(Field 7)

Pad An 8-byte pad area to align the I/O area on a double word boundary. No data is returned to this field.

Identifying free space

To retrieve the area name and the next available position within the sequential dependent part from all online areas, you can issue an unqualified POS call. This type of call also retrieves the unused space in the independent overflow and sequential dependent parts.

After a unsuccessful unqualified POS call, the I/O area contains the length (LL), followed by the same number of entries as existing areas within the database. Each entry contains the fields shown below:

Area Name
An 8-byte field giving the ddname from the AREA.

Position
An 8-byte field with binary zeros.

Unused SDEP CIs
A 4-byte field with binary zeros.

Unused IOV CIs
A 4-byte field with two binary zeros followed by a bad status code.

Commit-point processing in a DEDB

IMS retains database updates in processor storage until the program reaches a commit point. IMS saves updates to a DEDB in Fast Path buffers. The database updates are not applied to the DEDB until after the program has successfully completed commit-point processing.

Unlike Get calls to an MSDB, however, a Get call to an updated segment in a DEDB returns the updated value, even if a commit point has not occurred.

When a BMP is processing DEDBs, it must issue a `CHKP` or `SYNC` call to do commit-point processing before it terminates. Otherwise, the BMP abnormally terminates with abend U1008.

Related concepts:

"Commit-point processing in MSDBs and DEDBs" on page 327

P processing option

If the P processing option is specified in the PCB for your program, a GC status code is returned to your program whenever a call to retrieve or insert a segment causes a unit of work (UOW) boundary to be crossed.

Related reading: For more information on the UOW for DEDBs, see *IMS Version 11 Database Administration*.

Although crossing the UOW boundary probably has no particular significance for your program, the GC status code indicates that this is a good time to issue either a `SYNC` or `CHKP` call. The advantages of issuing a `SYNC` or `CHKP` call after your program receives a GC status code are:

- Your position in the database is retained. Issuing a `SYNC` or `CHKP` call normally causes position in the database to be lost, and the application program must reestablish position before it can resume processing.
- Commit points occur at regular intervals.

When a GC status code is returned, no data is retrieved or inserted. In your program, you can either:

- Issue a `SYNC` or `CHKP` call, and resume database processing by reissuing the call that caused the GC status code.
- Ignore the GC status code, and resume database processing by reissuing the call that caused the status code.

Related concepts:

"Calls with dependent segments for DEDBs" on page 337

H processing option

If the H processing option has been specified in the PCB for your call program, a GC status code is returned whenever a call to retrieve or insert a segment causes a unit of work (UOW) or an area boundary to be crossed. The program must cause a commit process before any other calls can be issued to that PCB.

If a commit process is not caused, an FR status code results (total buffer allocation exceeded), and all database changes for this synchronization interval are “washed” (sync-point failure).

A GC status code is returned when crossing the area boundary so that the application program can issue a SYNC or CHKP call to force cleanup of resources (such as buffers) that were obtained in processing the previous area. This cleanup might cause successive returns of a GC status code for a GN or GHN call, even if a SYNC or CHKP call is issued appropriately for the previous GC status code.

When an application is running HSSP and proceeding through the DEDB AREA sequentially, a buffer shortage condition may occur due to large IOV chains. In this case, a FW status code is returned to the application. Usually, the application issues a commit request and position is set to the next UOW. However, this does not allow the previous UOW to finish processing. In order to finish processing the previous UOW, you can issue a commit request after the FW status code is received and set the position to remain in the same UOW. You must also reposition the application to the position that gave the FW status code. The following shows an example of the command sequence and corresponding application responses.

```
GN                      root1
GN                      root2
GN                      root3
GN                      root4            /*FW status code received*/
CHKP
GN    SSA=(root4)       root4            /*User reposition prior to CHKP*/
GN                      root5
```

Calls with dependent segments for DEDBs

You can issue DL/I calls against direct and sequential dependent segments for DEDBs.

The DL/I calls that you can issue against a root segment are: GU, GN (GNP has no meaning for a root segment), DLET, ISRT, and REPL. You can issue all DL/I calls against a direct dependent segment, and you can issue Get and ISRT calls against sequential dependents segments.

Direct dependent segments

DL/I calls to direct dependents include the same number of SSAs as existing levels in the hierarchy (a maximum of 15). They can also include command codes and multiple qualification statements. The same rules apply to using command codes on DL/I calls to DEDBs as to full-function databases.

If you use the D command code in a call to a DEDB, the P processing option need not be specified in the PCB for the program. The P processing option has a different meaning for DEDBs than for full-function databases.

Some special command codes can be used only with DEDBs that use subset pointers. Your program uses these command codes to read and update the subset pointers.

Sequential dependent segments

Because sequential dependents are stored in chronological order, they are useful in journaling, data collection, and auditing application programs. You can access

sequential dependents directly. However, sequential dependents are normally retrieved sequentially using the Database Scan utility.

Restriction: When processing sequential dependent segments:

- You can only use the F command code with sequential dependents; IMS ignores all other command codes.
- You cannot use Boolean operators in calls to sequential dependents.

Related reading: For more information about the utility, see *IMS Version 11 Database Utilities*.

Related concepts:

"Processing Fast Path DEDBs with subset pointer command codes" on page 328

Related reference:

"P processing option" on page 336

DEDB DL/I calls to extract DEDB information

DL/I calls can be issued to obtain structural information about Data Entry Databases (DEDBs). Any application that can issue DL/I calls can take advantage of these DL/I calls.

There are two basic call types:

- The first type returns the minimum I/O area length required for a specific type '2' DL/I call.
- The second type returns specific information about the specified DEDB.

Each of these DL/I calls uses a call interface block called the Input Output Input Area (IOAI), a telecommunication program PCB (TP PCB), and specific calls that require an I/O area. Some required initialization of the IOAI is common for all calls and some initialization is specific to an individual call. The IOAI and the I/O area must be obtained in key 8 storage.

The following table describes the DL/I calls to extract DEDB information.

Table 56. DEDB DL/I Calls

DL/I Call	Description
AL_LEN	Returns the minimum length of the I/O area that is required for an AREALIST call.
DI_LEN	Returns the minimum length of the I/O area that is required for an DEDBINFO call.
DS_LEN	Returns the minimum length of the I/O area required for a DEDBSTR call.
AREALIST	Returns a list of areas that are part of the specified DEDB, with each area mapped by DBFCDAL1.
DEDBINFO	Returns DEDB information from the DMCB, mapped by DBFCDDI1.
DEDBSTR	Returns a list of segments and a segment data for DEDB with each segment mapped by DBFCDDS1.

The DL/I call that use the standard interface with register 1 must point to IOAI_CA.

The following figure shows the IOAI structure.

```
                                starting
                                 offset    note
IOAI_START     DS   0F
IOAI_NAME      DC   CL4'IOAI'      0      *1
IOAI_#FPU      DC   CL4'#FPU'      4      *1
IOAI_#FPI      DC   CL8'#FPUCDPI'  8      *1
IOAI_SUBC      DC   CL8' '        10      *1
*
IOAI_BLEN      DC   A(0)          18      *1
IOAI_ILEN      DC   A(0)          1C      *1
IOAI_IOAREA    DC   A(0)          20      *1
*
IOAI_CALL      DC   A(0)          24      *1
IOAI_PCBI      DC   A(0)          28      *1
IOAI_IOAI      DC   A(0)          2C      *1
*
IOAI_DLEN      DC   A(0)          30      *2
IOAI_STATUS    DC   CL2' '        34      *2
IOAI_B_LEVEL   DC   XL2'0'        36      *2
IOAI_STATUS_RC DC   A(0)          38      *2
IOAI_USERVER   DC   A(0)          3C      *1
IOAI_IMSVER    DC   A(0)          40      *2
*
IOAI_IMSLEVEL  DC   A(0)          44      *2
*
IOAI_APPL_NAME DC  CL8' '         48      *1
IOAI_USERDATA  DC  CL8' '         50      *1
IOAI_TIMESTAMP DC  CL8' '         58      *2
*                       input words.
IOAI_IN0       DC   A(0)          60      *3
IOAI_IN1       DC   A(0)          64      *3
IOAI_IN2       DC   A(0)          68      *3
IOAI_IN3       DC   A(0)          6C      *3
IOAI_IN4       DC   A(0)          70      *3
*                       feedback words
IOAI_FDBK0     DC   A(0)          74      *2
IOAI_FDBK1     DC   A(0)          78      *2
IOAI_FDBK2     DC   A(0)          7C      *2
IOAI_FDBK3     DC   A(0)          80      *2
IOAI_FDBK4     DC   A(0)          84      *2
*                       workareas.
IOAI_WA0       DC   A(0)          88      *4
IOAI_WA1       DC   A(0)          8C      *4
IOAI_WA2       DC   A(0)          90      *4
IOAI_WA3       DC   A(0)          94      *4
IOAI_WA4       DC   A(0)          98      *4
*
               DS 20F'0'          9C      for future expansion
IOAI_END_CHAR DC   CL4'IEND'      EC      *1
IOAI_LEN         len(DBFIOAI) = x'F0' bytes
```

Note:

1. The user is responsible for initializing these fields.
2. IMS uses these fields to return data to the caller. Which fields contain returned data depends on the DL/I call and are documented in the section on the specific call types.
3. May be used to pass additional data on the DL/I call, as documented under each DL/I call.
4. These fields are unchanged, and can be used as work areas by the application.

The fields in the following table must be initialized for all of the following DL/I calls.

Table 57. Field initialization for DEDB DL/I calls

Field	Description
IOAI_NAME	The characters 'IOAI' identifying this block.
IOAI_#FPU	The characters '#FPU' Indicating this is a #FPU call.
IOAI_#FPI	The characters '#FPUCDPI' indicating this is a subset call.
IOAI_SUBC	The DL/I call: AL_LEN, AREALIST, DS_LEN, DEDBSTR, DI_LEN or DEDBINFO.
IOAI_BLEN	The total length of the IOAI (x'F0').
IOAI_CALL	Address of IOAI_#FPU.
IOAI_PCBI	Address of the TPCB.
IOAI_IOAI	Address of this block. The user must set the high order bit on to indicate the end of the DL/I list.
IOAI_USERVER	Call version number. Defaults to one. This is the version number of a specific call. This field will be updated in the future if a specific call is altered such that the application must be sensitive to the changes.
IOAI_END_CHAR	The chars 'IEND' identifying the end of block.

The following fields are initialized for specific DL/I calls. If a specific call does not need an I/O area, these fields are ignored.

Table 58. Fields initialized for specific DEDB DL/I calls

Field	Description
IOAI_ILEN	The total length of the I/O area, including prefix and suffix.
IOAI_IOAREA	Address of the I/O area.
I/O Area	1st word: The I/O area length (same as IOAI_ILEN). Last word: X'FFFFFFFF', which is an 'end of I/O area' marker.
IOAI_IN0 -> IOAI_IN4	Five input words that might be required.

The following fields are updated by IMS for all the DEDB DL/I call types.

Table 59. Fields updated by IMS for all DL/I call types

Field	Description
IOAI_DLEN	The length of the output data that is returned by IMS. This field is informational only.
IOAI_STATUS	A 2-byte status code.
IOAI_STATUS_RC	A return code if needed.
IOAI_IMSVER	The maximum version of this call.

Table 59. Fields updated by IMS for all DL/I call types (continued)

Field	Description
IOAI_IMSLEVEL	The IMS level.

The following fields might be updated by specific DL/I calls.

Table 60. Fields updated by specific DL/I calls

Field	Description
I/O Area	1st word: unchanged. Data: see specific call types. Last word: potentially changed.
IOAI_FDBK0 -> IOAI_FDBK4	Five output words which may return data as documented by specific calls.

```
DBFCDAL1 mapping:        offset
CDAL_START   DS   0F
CDAL_ARNM    DS    CL8     00      Area name
CDAL_FLGS    DS    0XL4    08      Flag Bytes
CDAL_FLG1    DS    XL1     08      Flags for area status:
CDAL_F1OP     EQU   X'01'          - Area is opened
CDAL_F1BK     EQU   X'02'          - Temporary bit for backout
CDAL_F1UT     EQU   X'04'          - Utility active on this area
CDAL_F1ER     EQU   X'08'          - Error recovery needed
CDAL_F1AF     EQU   X'80'          - Sequential dep. part full
CDAL_F1EP     EQU   X'40'          - I/O error
CDAL_F1ST     EQU   X'20'          - Area stop request
CDAL_F1RE     EQU   X'10'          - Area restart request
CDAL_FLG2    DS    XL1     09       Reserved for Flag Byte #2
CDAL_FLG3    DS    XL1     0A       Reserved for Flag Byte #3
CDAL_FLG4    DS    XL1     0B       Reserved for Flag Byte #4
             DS    1F      0C       for growth
CDAL_LEND    DS    0F              End of area list entry
CDAL_LEN     EQU   *-&AA._START;   Len of area list entry

DBFCDDI0 mapping:        offset
CDDI_START   DS   0D
CDDI_DBNM    DS    CL8     00    Database name
CDDI_ANR     DS    H       08    Number of areas defined
CDDI_HSLV    DS    H       0A    Max SEGM level in the DB
CDDI_SGNR    DS    H       0C    Highest valid SEGM code
CDDI_SEGL    DS    H       0E    Maximum IOA length
CDDI_HBLK    DS    F       10    Number of anchor blocks
CDDI_RMNM    DS    CL8     14    Randomizing module name
CDDI_RMEP    DS    F       1C    Randomizing module entry point
             DS    8F      20    Reserved
             DS   0D             Align on double word boundary
CDDI_LEN     EQU   *-&AA._START;   Length of this area (x'40')

DBFCDDS1 mapping:         offset
CDDS_START   DS   0F
CDDS_GNAM    DS    CL8      00  SEGMENT NAME
CDDS_GDOF    DS    H        08  OFFSET FROM START SEQ TO DATA
CDDS_MAX     DS    H        0A  MAX SEG LEN
CDDS_MIN     DS    H        0C  MIN SEG LEN
CDDS_DBOF    DS    H        0E  OFFSET TO SEG ENTRIES
CDDS_NRFLD   DS    FL1      10  NUMBER OF FIELDS IN SEG
CDDS_SC      DS    FL1      11  SEGMENT CODE
CDDS_PREF    DS    H        12  POINTER OFFSET IN PARENT PREF
CDDS_FLG1    DS    X        14  FLAG BYTE
CDDS_FL1K     EQU   X'80'        KEY SEGMENT
CDDS_FL1S     EQU   X'40'        SEQUENTIAL DEP SEGMENT
CDDS_FL1P     EQU   X'20'        PCL POINTER TO PARENT
CDDS_FISRT   DS    X        15  INSERT RULES
CDDS_PARA    DS    H        16  OFFSET TO PARENT SEGMENT
```

```
CDDS_SBLP    DS    F        18  SIBLING POINTER
CDDS_LEVL    DS    XL1      1C  SEGMENT LEVEL
CDDS_KEYL    DS    XL1      1D  KEY LENGTH - 1
CDDS_KDOF    DS    H        1E  OFFSET TO KEY FIELD IN SEGMENT
CDDS_RSRVE   DS    XL4      20  FOR USE IN UMDR0 | RESERVED
CDDS_CMPC    DS    A        24  A(CMPC)
CDDS_FLG2    DS    XL1      28  FLAG BYTE 2 (fixed length)
             DS    XL3      29  FOR GROWTH
             DS   5F        2C  for growth
CDDS_END     DS   0F            END
CDDS_LEN     EQU    *-&AA._START;  len of SDB entry
```

The following status codes are specific to these new DL/I calls.

Table 61. Status codes for specific DEDB DL/I calls

Status Code	Description
AA	Invalid #FPU/#FPUCDPI call.
AB	Getmain error.
AC	DEDB name not found.
AD	The I/O area was not long enough to contain the data.
AE	IOAI_LEN was zeros. It must be filled by the caller.
AF	The I/O area address was not passed in by IOAI_IOAREA.
AG	The IOAI does not point to itself, IOAI_IOAI.
AH	The IOAI did not contain 'IOAI'.
AI	The I/O area length in the I/O area does not match IOAI.
AJ	The I/O area did not contain the end-of-list marker.
AK	The IOAI did not have end-of-block marker 'IEND'.
AL	IOAI_BLEN is not correct.
AM	DEDB not passed in via the IOAI on the #FPUCDPI call.

AL_LEN Call

The AL_LEN call returns the minimum length of the I/O area required for an AREALIST call.

Input

IOAI
Formatted and filled out as documented above.

IOAI_IN0
Points to storage containing the DEB name.

Output

IOAI_STATUS
Call status, ' ' means successful.

IOAI_FDBK0
The minimum length of the I/O area.

IOAI_FDBK1
The number of AREAS in this DEDB.

DI_LEN Call

Return the minimum length of the I/O area required for an DEDBINFO call.

Input

IOAI
Formatted and filled out as documented above.

IOAI_IN0
Points to storage containing the DEB name.

Output

IOAI_STATUS
Call status, ' ' means successful.

IOAI_FDBK0
The minimum length of the I/O area.

DS_LEN Call

Return the minimum length of the I/O area required for a DEDBSTR call.

Input

IOAI
Formatted and filled out as documented above.

IOAI_IN0
Points to storage containing the DEB name.

Output

IOAI_STATUS
Call status, ' ' means successful.

IOAI_FDBK0
The minimum length of the I/O area.

IOAI_FDBK1
The number of SEGMENTS in this DEDB.

AREALIST Call

The AREALIST call returns a list of areas that are part of the specified DEDB, with each area mapped by DBFCDAL1.

Input

IOAI
Formatted and filled out as documented above.

IOAI_IN0
Points to storage containing the DEB name.

I/O Area
Formatted as documented above.

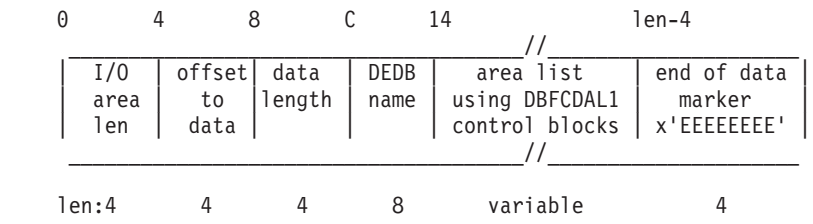
Output

IOAI_STATUS
Call status, ' ' means successful.

IOAI_FDBK0
The minimum length of the I/O area.

IOAI_FDBK1
The number of AREAS in this DEDB.

The I/O Area



DEDBINFO Call

Return DEDB information from the DMCB, mapped by DBFCDDI1.

Input

IOAI
Formatted and filled out as documented above.

IOAI_IN0
Points to storage containing the DEB name.

I/O Area
Formatted with length in the first word, and 'FFFFFFFF' as an end of I/O area marker.

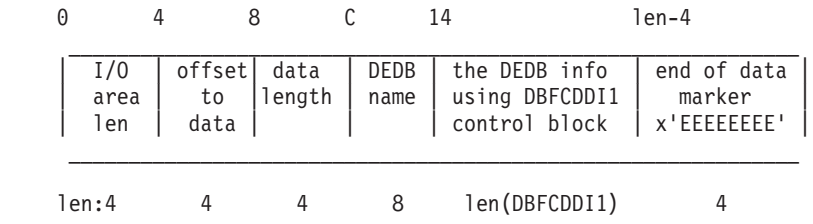
Output

IOAI_FDBK0
The minimum length of the I/O area.

IOAI_FDBK1
The minimum I/O area for the DEDBSTR call.

IOAI_FDBK2
The minimum I/O area for the AREALIST call.

The I/O Area



DEDSTR Call

Return a list of segments and segment data for a DEDB with each segment mapped by DBFCDDS1.

Input

IOAI
Formatted and filled out as documented above.

IOAI_IN0
Points to storage containing the DEB name.

I/O Area
Formatted with length in the first word, and 'FFFFFFFF' as an end of I/O area marker.

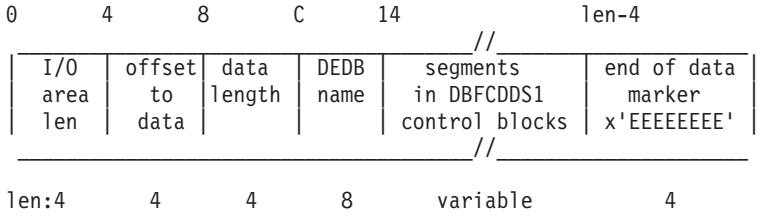
Output

IOAI_STATUS
The minimum length of the I/O area.

IOAI_FDBK0
The minimum I/O area for the DEDBSTR call.

IOAI_FDBK1
The minimum I/O area for the SEGMENTS call.

The I/O Area



Fast Path coding considerations

You can use DL/I calls to access Fast Path databases. You can also use two additional calls: FLD and POS. The type of Fast Path database that you are processing determines when you can use each of these calls.

To process MSDBs, you can use these calls:

- For nonterminal-related MSDBs:
  - FLD
  - GU and GHU
  - GN and GHN
  - REPL
- For terminal-related, fixed MSDBs:
  - FLD
  - GU and GHU
  - GN and GHN
  - REPL
- For terminal-related, dynamic MSDBs:
  - DLET
  - FLD
  - GU and GHU
  - GN and GHN
  - ISRT

REPL

You can use these calls to process a DEDB:

- DEQ
- DLET
- FLD
- GU and GHU
- GN and GHN
- GNP and GHNP
- ISRT
- POS
- REPL
- RLSE

Chapter 21. Writing ODBA application programs

By using the ODBA interface, IMS DB databases can be accessed from environments that are outside the scope of control for IMS, such as DB2 for z/OS stored procedures.

The ODBA interface is not needed within IMS-controlled regions, such as MPRs, BMPs, or IFPs, for calls to locally controlled databases.

The z/OS application programs (hereafter called the ODBA application programs) run in a separate z/OS address space that IMS regards as a separate region from the control region. The separate z/OS address space hereafter is called the z/OS application region.

The ODBA interface gains access to IMS DB through the Database Resource Adapter (DRA). The ODBA application programs (which can access any address space within the z/OS they are running in) gain access to IMS DB databases through the ODBA interface. The following figure illustrates this concept and shows the relationship between the components of this environment.

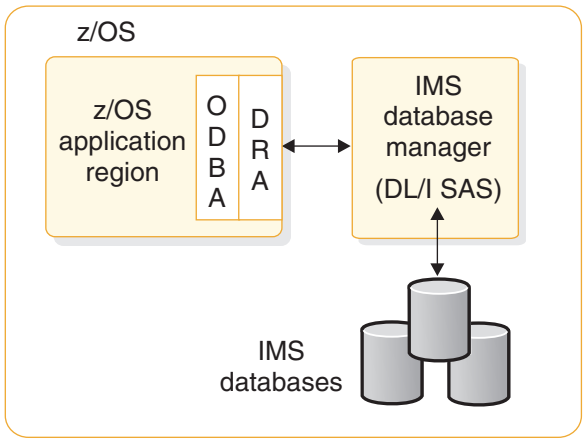


Figure 69. z/OS application region's connection to IMS DB

One z/OS application region can connect to multiple IMS DBs and multiple z/OS application regions can connect to a single IMS DB. The connection is similar to that of CICS to DBCTL.

General application program flow of ODBA application programs

An ODBA application program issues calls, including DL/I calls, using the AERTDLI interface of an application interface block (AIB).

Several conditions must be met for the AIB call to succeed:

1. If an AIB is not passed in the call, a U261 abend is issued.
2. If the AIB that is passed is not valid, a U476 abend is issued.
3. If the AIB that is passed is not large enough (264 bytes), the AIB return and reason codes are set to X'104' and X'228'.

© Copyright IBM Corp. 1974, 2015

4. If the AIB that is passed is not on a fullword boundary, the z/OS system will return an abend S201.
5. If there are other internal problems with the call, other return and reason codes are passed back to the z/OS application program. See *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes* for a complete list of these return and reason codes.

The ODBA application program must link edit with a language module (DFSCDLI0) or this module can be loaded into the z/OS application region. The entry point for DFSCDLI0 is AERTDLI.

Restriction: The ODBA interface does not support calls into batch DL/I regions.

The basic program flow for an ODBA application program is:

1. Establish the application execution environment. The application execution environment must be initialized in the z/OS application region.

   To initialize the environment, use either the CIMS INIT call or, if you need to establish connections to multiple IMS systems, the CIMS CONNECT call.

   If you use the INIT subfunction of CIMS you can include a startup table ID in the optional AIBRSNM2 field of the AIB to connect to the IMS DB system listed in the startup table. If the AIBRSNM2 field is blank, connect to the IMS DB when you allocate a PSB.

   The form of the connection call is:

   ```
   CALL AERTDLI parmcount, CIMS, AIB
   ```

   Where:

   CIMS Is the required call function.

   AIB Has the following fields:

   AIBSFUNC
   The subfunction is either INIT or CONNECT. This field is mandatory.

   AIBRSNM1
   An optional field that provides an eye catcher identifier of the application server that is associated with the AIB. This field is 8 bytes.

   AIBRSNM2
   An optional field for the INIT subfunction in which you can specify an optional 4-byte startup table ID. The ID is optional if the call is issued as preconditioning only. If the ID is given, the z/OS application region connects to the IMS DB specified in the DBCTLID parameter of the selected startup table. The AIBRSNM2 field is not supported with the CONNECT subfunction.

   The characteristics of the connection are determined from the DRA startup table. The startup table name is DFSxxxx0, where xxxx is the startup table ID that is used in the CIMS and APSB calls. Each startup table defines a combination of connection attributes, one of which is a subsystem ID of the IMS DB.

   If you use the CONNECT subfunction, the calling application program can optionally supply its own connection properties table by specifying the address

of the table in an entry in the ODBA data store connection table used by the CONNECT subfunction. The connection properties table is mapped by the DFSPRP macro.

Related Reading: For more information about building a DRA startup table, see *IMS Version 11 System Definition*.

2. Allocate a PSB. The APSB call, introduced for CPIC-driven programs, is used with the ODBA interface to allocate a PSB for the z/OS application region.

   Security is checked before the call can succeed. For more information, see "Accessing IMS databases through the ODBA interface" in *IMS Version 11 Communications and Connections*.

   The APSB call is in the following form:

   ```
   CALL AERTDLI parmcount, APSB, AIB
   ```

   Where:

   APSB Is the required call function.

   AIB Is the name of the application interface block. The fields in the AIB must be filled in:

   AIBRSNM1
   Is the 8-character PSB name.

   AIBRSNM2
   Is the 4-byte IMS alias name that is used as the startup table ID.

   Several conditions must be met for the allocation request to succeed.

   - The PSB must exist and security checking through RACF must succeed.
   - An ODBA environment must have been established by either a CIMS INIT call or a CIMS CONNECT call.
   - z/OS Resource Recovery Services (RRS) must be active when the APSB call is made.

   Multiple PSBs can be active at the same time, which is typical for server environments. No token is specifically provided to identify which PSB is to be used for a given call to a given IMS DB, so the same AIB *must* be used for all calls to the same PSB instance (APSB, DB calls, DPSB). This enables multiple instances of the same PSB to be in use for the same IMS DB at the same time. The parallelism is controlled by the thread count specified in the startup table. The maximum number of threads and dependent regions supported by an IMS DB instance is 999.

3. Perform DB calls. All DL/I calls, with a few exceptions, are supported through the AIB. The unsupported calls entail message handling (the IOPCB is available only for system calls), CKPT, ROLL, ROLB, and INQY PROGRAM. Alternate destination PCBs cannot be used. Both full-function databases and DEDBs are available.
4. Commit the changes. Synchronization is performed by issuing the distributed commit calls, SRRCMIT or ATRCMIT, or possibly their rollback forms of SRRBACK or ATRBACK. IMS sync-point calls are not allowed. Commit is effective for all RRS controlled resources in the z/OS task.
5. Deallocate the PSB.

   The DPSB call is used when the work unit is complete. In the default case, a commit call must be issued before a DPSB call can be issued. No DL/I call, including system service calls, can be made between the commit and the DPSB call.

   The DPSB call is in the following form:

```
CALL AERTDLI parmcount, DPSB, AIB
```

Where:

DPSB Is the required call function.

AIB Is the name of the application interface block. The following fields in the AIB must be filled in:

AIBRSNM1
Is the 8-character PSB name.

AIBSFUNC
Is an optional field. Set it to 'PREPbbbb' when you want to deallocate the PSB before initialization of commit processing and when the commit processing is provided from outside the application.

IMS performs phase 1 commit processing and returns control to the requestor, but holds the in-doubt work until RRS (the commit manager) requests full commit processing. An example is in DB2 UDB for z/OS Stored Procedures, whereDB2 for z/OS initializes commit processing on behalf of the procedure. See DB2 for z/OS for a discussion of this scenario.

6. Terminate the connection.

   The termination call is in the following form:

   ```
   CALL AERTDLI parmcount, CIMS, AIB
   ```

   Where:

   CIMS Is the required call function.

   AIB Is the name of the application interface block. The following fields in the AIB must be filled in:

   AIBSFUNC
   Is a mandatory field whose value is TERM or TALL. Use TERM to sever a single IMS DB connection. Use TALL to sever all connections for this z/OS application region and remove the DRA from the address space.

   AIBRSNM1
   Is an optional field that provides an eye catcher identifier of the application server associated with the AIB. This field is 8 bytes in length.

   AIBRSNM2
   When subfunction equals TERM, provides the 4-byte startup table ID used in a previous APSB call. This field is not needed when the subfunction equals TALL.

Server program structure

The commit scope within the z/OS application environment is all the work under the TCB from which the commit request is made to z/OS Resource Recovery Services (RRS). Server environments, therefore, need a separate TCB under which the individual client requests will be managed. Each TCB will map to a PST for thread handling.

The following figure shows an example TCB structure for a server environment.

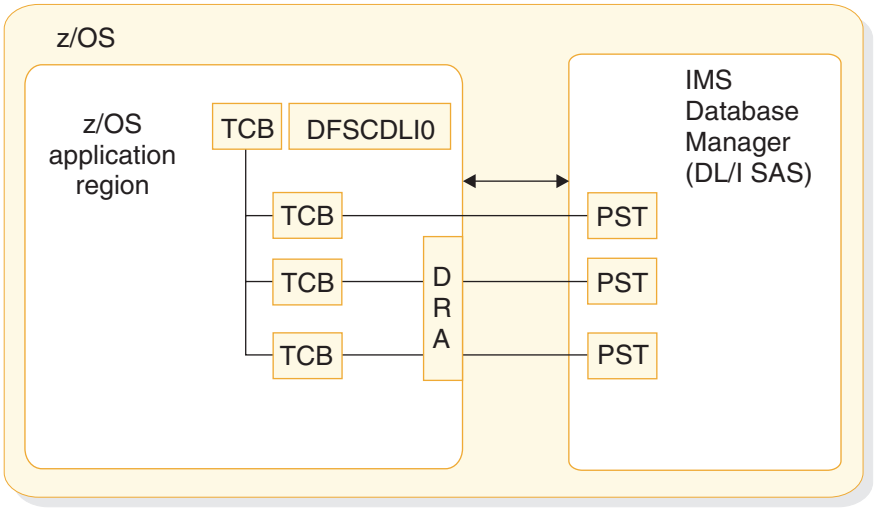


Figure 70. DRA uses one TCB per thread

Each connection to an IMS DB uses a thread under the TCB. When the APSB call is processed, a context is established and tied to the TCB. At commit time, all contexts for this TCB are committed or aborted by RRS.

Loading DFSCDLI0 rather than link editing is attractive when the z/OS application region is a server supporting many clients with many instances of threads connected with the IMS DBs.

DB2 for z/OS stored procedures use of ODBA

DB2 for z/OS stored procedures connecting to ODBA must run in a z/OS Workload Manager-managed (WLM-managed) stored procedures address space.

DB2 for z/OS establishes the ODBA environment by specifying either the INIT subfunction or the CONNECT subfunction of the CIMS call for the stored procedure address space. If the CIMS INIT call is issued, the connection to a specific IMS DB occurs when the APSB call is issued. If the CIMS CONNECT call is used, the connection to one or more IMS DB systems can optionally occur either when the CIMS CONNECT call is issued or when the APSB call is issued.

Each stored procedure running in the stored procedure address space runs under its own TCB that is established by DB2 for z/OS when the stored procedure is initialized. DB2 for z/OS issues the commit call on behalf of the stored procedure when control is returned to DB2 for z/OS. Only the PREP subfunction of the DPSB call should be issued by the stored procedures.

Restriction: If stored procedures are nested under a single WLM stored procedure address space and call IMS ODBA, the ODBA threads will hang.

The following figure illustrates the connection from a DB2 for z/OS stored procedures address space to an IMS DB subsystem. This connection allows DL/I data to be presented through an SQL interface, either locally to this DB2 for z/OS or to DRDA® connected DB2 for z/OS databases.

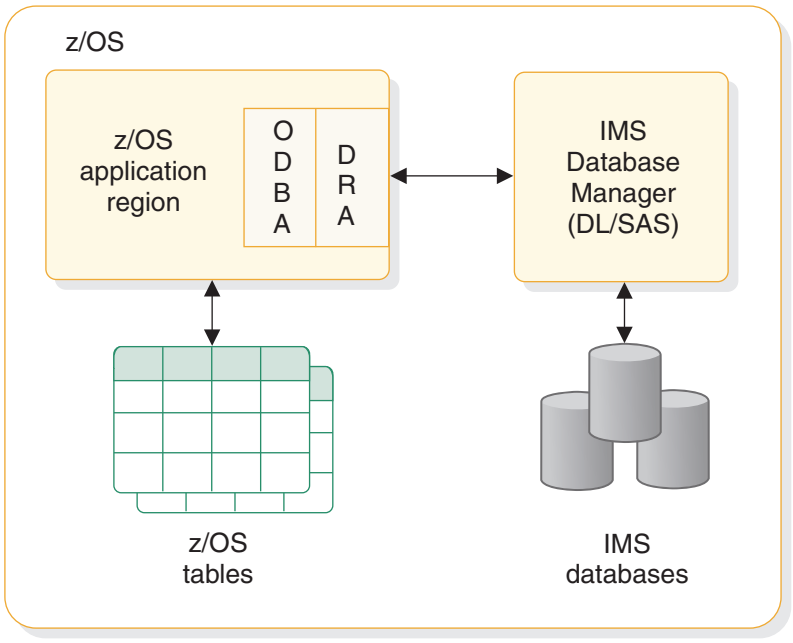


Figure 71. DB2 for z/OS stored procedures connection to IMS DB

The following figure illustrates the general relationships involved with using DB2 for z/OS stored procedures and IMS DB together.

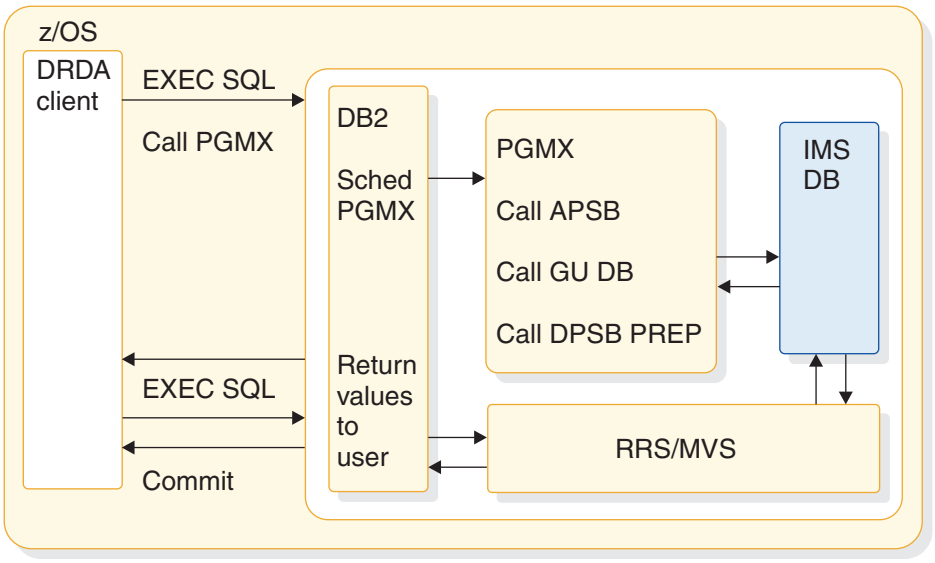


Figure 72. DB2 for z/OS stored procedures relationships

Testing an ODBA application program

You should perform a program unit test on your ODBA application program to ensure that the program correctly handles its input data, processing, and output data. The amount and type of testing you do depends on the individual program.

Be aware of your established test procedures before you start to test your program. To begin testing, you need the following items:

- A test JCL statement

- A test database

  Always begin testing programs against test-only databases. Do not test programs against production databases. If the program is faulty it might damage or delete critical data.
- Test input data

  The input data that you use need not be current, but it should be valid data. You cannot be sure that your output data is valid unless you use valid input data.

The purpose of testing the program is to make sure that the program can correctly handle all the situations that it might encounter. To thoroughly test the program, try to test as many of the paths that the program can take as possible. For example:

Test each path in the program by using input data that forces the program to execute each of its branches. Be sure that your program tests its error routines. Again, use input data that will force the program to test as many error conditions as possible. Test the editing routines your program uses. Give the program as many different data combinations as possible to make sure it correctly edits its input data. The following table lists the tools you can use to test Online (IMSDB), Batch, and BMP programs.

Table 62. Tools you can use for testing your program

Tool	Online (IMS DB)	Batch	BMP
DFSDDLT0	No	Yes[1]	Yes
DL/I image capture program	Yes	Yes	Yes

Note: 1. For call-level programs only. (For a command-level batch program, you can use DL/I image capture program first, to produce calls for DFSDDLT0).

Tracing DL/I calls with image capture to test your ODBA program

The DL/I image capture program (DFSDLTR0) is a trace program that can trace and record DL/I calls issued by batch, BMP, and online (IMS DB environment) programs. You can produce calls for use as input to DFSDDLT0.

You can use the image capture program to:

- Test your program

  If the image capture program detects an error in a call it traces, it reproduces as much of the call as possible, although it cannot document where the error occurred, and cannot always reproduce the full SSA.
- Produce input for DFSDDLT0 (DL/I test program)

  You can use the output produced by the image capture program as input to DFSDDLT0. The image capture program produces status statements, comment statements, call statements, and compare statements for DFSDDLT0. For example, you can use the image capture program with a ODBA application, to produce calls for DFSDDLT0.
- Debug your program

  When your program terminates abnormally, you can rerun the program using the image capture program. The image capture program can then reproduce and

document the conditions that led to the program failure. You can use the information in the report produced by the image capture program to find and fix the problem.

Using image capture with DFSDDLT0 to test your ODBA program

The image capture program produces the following control statements that you can use as input to DFSDDLT0.

- Status statements

  When you invoke the image capture program, it produces the status statement. The status statement it produces:

  – Sets print options so that DFSDDLT0 prints all call trace comments, all DL/I calls, and the results of all comparisons

  – Determines the new relative PCB number each time a PCB change occurs while the application program is running

- Comments statement

  The image capture program also produces a comments statement when you run it. The comments statements give:

  The time and date IMS started the trace

  The name of the PSB being traced

  The image capture program also produces a comments statement preceding any call in which IMS finds an error.

- Call statements

  The image capture program produces a call statement for each DL/I call.

- Compare statements

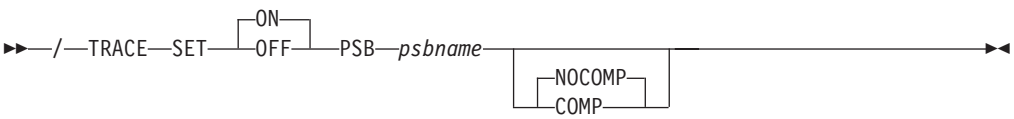
  If you specify COMP on the DLITRACE control statement, the image capture program produces data and PCB comparison statements.

Running image capture online

When you run the image capture program online, the trace output goes to the IMS log data set. To run the image capture program online, you issue the IMS TRACE command from the z/OS console.

If you trace a BMP and you want to use the trace results with DFSDDLT0, the BMP must have exclusive write access to the databases it processes. If the application program does not have exclusive access, the results of DFSDDLT0 may differ from the results of the application program.

The following diagram shows TRACE command format:



SET ON | OFF
Turns the trace on or off.

PSB *psbname*
Specifies the name of the PSB you want to trace. You can trace more than one PSB at the same time by issuing a separate TRACE command for each PSB.

COMP | NOCOMP
Specifies whether you want the image capture program to produce data and PCB compare statements to be used with DFSDDLT0.

Retrieving image capture data from the log data set

If the trace output is sent to the IMS log data set, you can retrieve it by using utility DFSERA10 and a DL/I call trace exit routine, DFSERA50.

DFSERA50 deblocks, formats, and numbers the image capture program records to be retrieved. To use DFSERA50, you must insert a DD statement defining a sequential output data set in the DFSERA10 input stream. The default ddname for this DD statement is TRCPUNCH. The card must specify BLKSIZE=80.

Examples: You can use the following examples of DFSERA10 input control statements in the SYSIN data set to retrieve the image capture program data from the log data set:

- Print all image capture program records:

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,FLDTYP=X
```

- Print selected image capture program records by PSB name:

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT OFFSET=25,VLDTYP=C,FLDLEN=8,
                   VALUE=psbname, COND=E
```

- Format image capture program records (in a format that can be used as input to DFSDDLT0):

```
Column 1       Column 10
OPTION         PRINT OFFSET=5,VALUE=5F,COND=M
OPTION         PRINT EXITR=DFSERA50,OFFSET=25,FLDTYP=C
               VALUE=psbname,FLDLEN=8,DDNAME=OUTDDN,COND=E
```

  The DDNAME= parameter is used to name the DD statement used by DFSERA50. The data set defined on the OUTDDN DD statement is used instead of the default TRCPUNCH DD statement. For this example, the DD appears as:

```
//OUTDDN DD ...,DCB=(BLKSIZE=80),...
```

Requests for monitoring and debugging your ODBA program

To debug your ODBA program, you can issue the statistics (STAT) or log (LOG) request.

You can use the following two requests to help you in debugging your program:

- The statistics (STAT) request retrieves database statistics. STAT can be issued from both call- and command-level programs.
- The log (LOG) request makes it possible for the application program to write a record on the system log. You can issue LOG as a command or call in a batch program; in this case, the record is written to the IMS log. You can issue LOG as a call or command in an online program in the IMS DB environment; in this case, the record is written to the IMS DB log.

What to do when your ODBA program terminates abnormally

Whenever your program terminates abnormally, you can take some actions to simplify the task of finding and fixing the problem. ODBA does not issue any return or reason codes. Most non-terminating errors for ODBA application programs are communicated in AIB return and reason codes. You can record as much information as possible about the circumstances under which the program terminated abnormally. In addition, you can check for certain initialization and execution errors.

Recommended actions after an abnormal termination of an ODBA program

The suggestions given here are some common guidelines on what you should do if your program terminates abnormally.

- Document the error situation to help in investigating and correcting it. Some of the information that can be helpful include:
  - The program's PSB name
  - The call function
  - The terminal ID (online programs only)
  - The contents of the AIB or the PCB
  - The contents of the I/O area when the problem occurred
  - If a database request was executing, the SSAs or SEGMENT and WHERE options, if any, the request used
  - The date and time of day
- When your program encounters an error, it can pass all the required error information to a standard error routine.
- You can send a message to the system log by issuing a LOG request.

Diagnosing an abnormal termination of an ODBA program

If your program does not run correctly when you are testing it or when it is running, you need to isolate the problem. The problem might be anything from a programming error (for example, an error in the way you coded one of your requests) to a system problem.

You can check for the following errors when your program fails to run, terminates abnormally, or gives incorrect results.

ODBA initialization errors

Before your program receives control, IMS must have correctly loaded and initialized the PSB and DBDs used by your application program. Often, when the problem is in this area, you need a system programmer or DBA (or your equivalent specialist) to fix the problem. One thing you can do is to find out if there have been any recent changes to the DBDs, PSB, and the control blocks that they generate.

ODBA running errors

If you do not have any initialization errors, check the following in your program:

1. The output from the compiler. Make sure that all error messages have been resolved.
2. The output from the binder:
   - Are all external references resolved?

- Have all necessary modules been included?
- Was the language interface module correctly included?

3. Your JCL. Is the information that described the files that contain the databases correct? If not, check with your DBA.

Chapter 22. Programming with the IMS support for DRDA

IMS provides an implementation of the Distributed Relational Database Architecture™ (DRDA) protocol that you can use to write your own IMS Connect TCP/IP client applications.

DRDA is an open architecture that enables communication between applications and database systems on disparate platforms. Details about using the DRDA protocol to perform database access operations are in the open specifications for DRDA. The following information describes only the IMS-specific extensions provided by the IMS support for DRDA.

To use the IMS support for DRDA, you must create the DRDA client driver (DRDA source server). No additional software needs to be installed or configured on the client system. The DRDA target server consists of IMS Connect and the Open Database Manager (ODBM) running with IMS in z/OS.

The IMS support for DRDA includes support for both application-directed transaction demarcation (local) and XA-enabled (global) transactions.

IMS does not support the following DRDA functions:

- Multi-row input
- Client reroute
- Security plugin

The IMS support for DRDA is based on the DRDA Version 4 technical standard. The DRDA specification is documented by the Open Group Consortium at www.opengroup.org.

Server compatibility checking

All communication between a source and target DRDA server begins with initialization and security. In the initialization flow, the DRDA client issues the EXCSAT command and an EXCSATRD data object is sent back from the DRDA target server.

In the IMS support for DRDA implementation, the EXCSATRD reply data object includes a Server Release Level (SRVRLSLV) parameter. The SRVRLSLV parameter is a string that specifies the version number of the distributed database management (DDM) language recognized by the IMS Connect and ODBM server components. This string is used by the client to perform server compatibility checking to ensure that both IMS Connect and ODBM understand any codepoints that the client sends. The DDM version numbering is specific to the IMS support for DRDA. All compatibility checking for the IMS support for DRDA is done based on the SRVRLSLVL parameter.

Important: The SRVRLSLV parameter value sent back from the target server in response to the EXCSAT command is OD-ICON 1 OD-ODBM 1.

Updating the source server with the latest maintenance release of IMS without applying the same maintenance release to all your IMS Connect or ODBM installations may cause the source server to be out of synchronization with the

© Copyright IBM Corp. 1974, 2015

target server. To prevent this possibility, the server compatibility check allows the connection to be made only if the IMS support for DRDA target server recognizes the DDM version level that is used by the source server.

How IMS data is mapped to the DRDA protocol

In a database query operation with the IMS support for DRDA, a *row* is defined as the concatenation of an instance of the aibdbpcbStream data structure plus all of the requested fields within an IMS hierarchic path. An aibdbpcbStream instance is a concatenation of an instance of the aibStream data structure followed by an instance of the dbpcbStream data structure. The requested fields are represented by the RTRVFLD objects sent with an OPNQRY command. The concatenation of the aibdbpcbStream instance and data fields represents a single row in a query row set.

The IMS support for DRDA supports only flexible blocking, where each query block can be a different size, depending on the size of the row or result set being returned. The specified query block size is used as an initial size, and the query block can expand beyond that size, if necessary, to complete the fetch operation.

In the IMS support for DRDA implementation, data is returned from the DRDA target server in byte stream format, and the client is responsible for data type processing.

Related concepts:

Overview of the CSL Open Database Manager (System Administration)

IMS Connect support for access to IMS DB (Communications and Connections)

Related reference:

DRDA DDM command architecture reference (Application Programming APIs)

DDM commands for data operations with the IMS support for DRDA

Use the distributed database management (DDM) commands provided by the IMS support for DRDA for singleton and batch data operations.

Before accessing the database, you need to first establish a database connection by issuing an ACCRDB command from your DRDA client application and successfully receive an ACCRDBRM data object back from the DRDA target server.

After the connection is established, you can issue DDM commands to access data from your DRDA client application.

- To retrieve data, issue an OPNQRY command.
- To insert, update, or delete data, issue an EXCSQLIMM command.

Data operations can be in singleton or batch operations. Specify the type of data operation by setting the Byte String Data Representation (BYTSTRDR) parameter in the DLIFUNC command object that is chained to the DDM command.

The following table shows the DDM commands that the DRDA client issues for data operations with the IMS support for DRDA.

Table 63. DDM commands for data operations with the IMS support for DRDA

Data operation	DDM command	**BYTSTRDR** parameter value for DLIFUNC **command object**
Insert data	EXCSQLIMM	ISRT
Retrieve data - DL/I Get Hold Unique	OPNQRY	GHU
Retrieve data - DL/I Get Unique	OPNQRY	GU
Retrieve data - DL/I Get Hold Next	OPNQRY	GHN
Retrieve data - DL/I Get Next	OPNQRY	GN
Retrieve data - DL/I Get Hold Next Within Parent	OPNQRY	GHNP
Retrieve data - DL/I Get Next Within Parent	OPNQRY	GNP
Update data	EXCSQLIMM	REPL
Delete data	EXCSQLIMM	DLET

The following table shows the DDM commands that the DRDA client issues for batch data operations with the IMS support for DRDA.

Table 64. DDM commands for batch data operations with the IMS support for DRDA

Batch data operation	DDM command	**BYTSTRDR** parameter value for DLIFUNC **command object**
Retrieve data	OPNQRY	RETRIEVE
Update data	EXCSQLIMM	UPDATE
Delete data	EXCSQLIMM	DELETE

Related reference:

DRDA DDM command architecture reference (Application Programming APIs)

Part 3. Application programming for IMS TM

IMS provides support for writing application programs to access IMS transactions.

© Copyright IBM Corp. 1974, 2015

Chapter 23. Defining application program elements for IMS TM

You can write application programs to communicate with the IMS Transaction Manager using DL/I calls in assembler language, C, COBOL, Java, Pascal, or PL/I.

Formatting DL/I calls for language interfaces

When you use DL/I calls in assembler language, C language, COBOL, Pascal, or PL/I, you must call the DL/I language interface to initiate the functions specified with the DL/I calls.

IMS offers several interfaces for DL/I calls:

- A language-independent interface for any programs that are Language Environment conforming (CEETDLI)
- Language-specific interfaces for all supported languages (xxxTDLI)
- A non-language-specific interface for all supported languages (AIBTDLI)

Java makes use of the all three DL/I language interfaces, but the usage is internal and no calls are necessary to initiate the functions specified with the DL/I calls.

Related concepts:

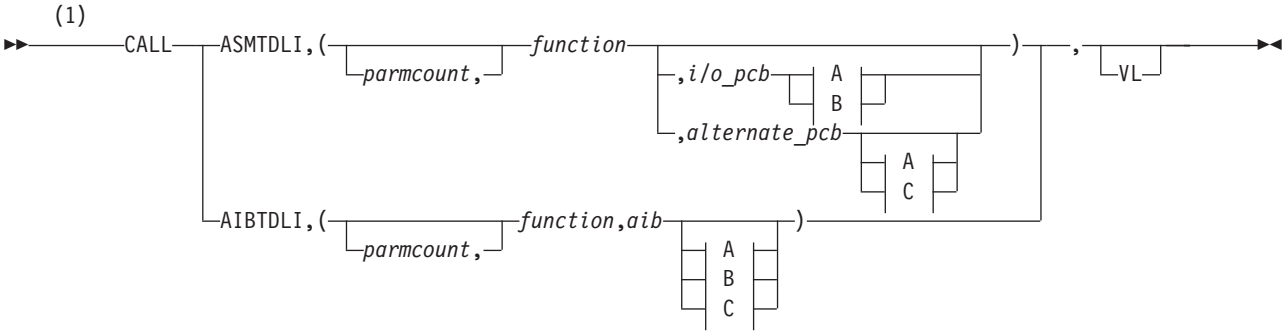
Chapter 35, "IMS solutions for Java development overview," on page 539

Application programming for assembler language

Application programs in assembly language use the following format, parameters, and DL/I calls to communicate with the IMS Transaction Manager.

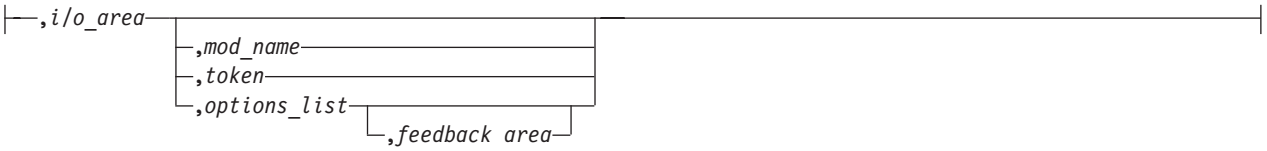
In assembler language programs, all DL/I call parameters that are passed as addresses can be passed in a register, which, if used, must be enclosed in parentheses.

Format

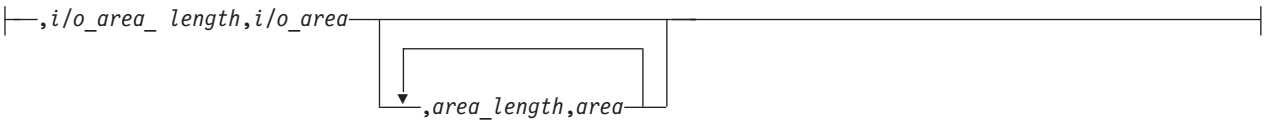


© Copyright IBM Corp. 1974, 2015

A:



B:



C:

Notes:

1 Assembler language programs must use either *parmcount* or VL.

Parameters

parmcount
Specifies the address of a 4-byte field in user-defined storage that contains the number of parameters in the parameter list that follows *parmcount*. Assembler language application programs must use either *parmcount* or VL.

function
Specifies the address of a 4-byte field in user-defined storage that contains the call function to be used. The call function must be left-justified and padded with blanks. For example, (GUbb is a call function.

i/o pcb
Specifies the address of the I/O program communication block (PCB). The I/O PCB address is the first address passed on entry to the application program in the PCB list, given the following circumstances:

- A program executing in DLI or database management batch (DBB) regions where `CMPAT=YES` is coded on the PSB.
- Any program executing in batch message processing program (BMP), message processing program (MPP), or IMS Fast Path (IFP) regions regardless of the `CMPAT=` value.

alternate pcb
Specifies the address of the alternate PCB to be used for the call. The PCB address must be one of the PCB addresses passed on entry to the application program in the PCB list.

aib
Specifies the address of the application interface block (AIB) in user-defined storage.

i/o area
Specifies the address of the I/O area in user-defined storage used for the call. The I/O area must be large enough to contain the returned data.

i/o area length
Specifies the address of a 4-byte field in user-defined storage that contains the I/O area length (specified in binary).

area length
Specifies the address of a 4-byte field in user-defined storage that contains the length (specified in binary) of the area immediately following it in the parameter list. Up to seven area length/area pairs can be specified.

area
Specifies the address of the area in user-defined storage to be checkpointed. Up to seven area length/area pairs can be specified.

token
Specifies the address of a 4-byte field in user-defined storage that contains a user token.

options list
Specifies the address of the *options list* in user-defined storage that contains processing options used with the call.

feedback area
Specifies the address of the feedback area in user-defined storage that receives information about options list processing errors.

mod name
Specifies the address of an 8-byte area in user-defined storage that contains the user-defined MOD name used with the call. The *mod name* parameter is used only with MFS.

destination name
Specifies the address of an 8-byte field in user-defined storage that contains the name of the logical terminal or transaction code to which messages resulting from the call are sent.

VL Signifies the end of the parameter list. Assembler language programs must use either *parmcount* or VL.

Example DL/I call formats

DL/I AIBTDLI interface:

```
CALL  AIBTDLI,(function,aib,i/o area),VL
```

DL/I language-specific interface:

```
CALL  ASMTDLI,(function,i/o pcb,i/o area),VL
```

Related concepts:

"AIBTDLI interface" on page 245

Related reference:

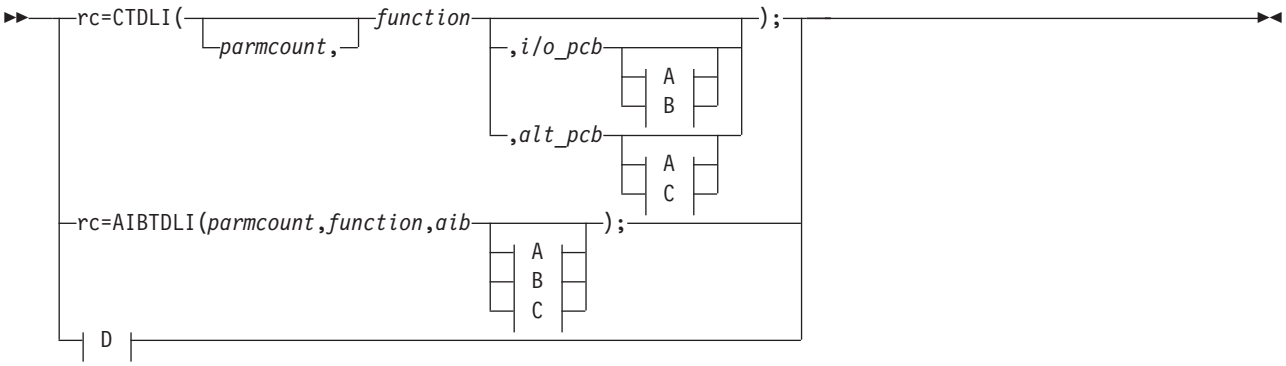
DL/I calls for transaction management (Application Programming APIs)

DL/I calls for IMS TM system services (Application Programming APIs)

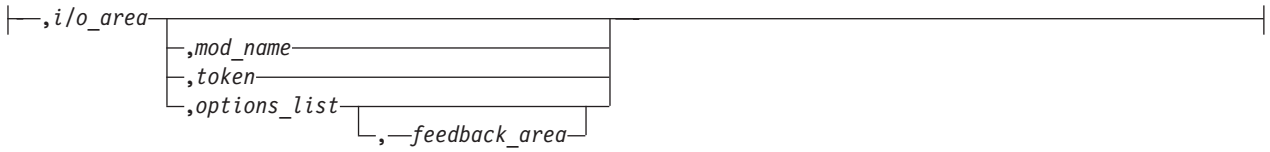
Application programming for C language

Application programs in C use the following format, parameters, and DL/I calls to communicate with the IMS Transaction Manager.

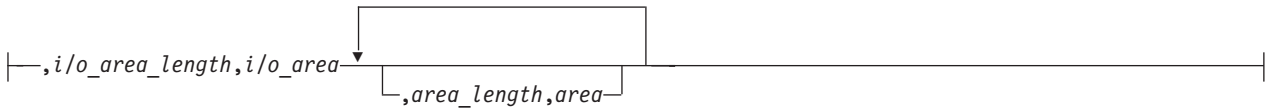
Format



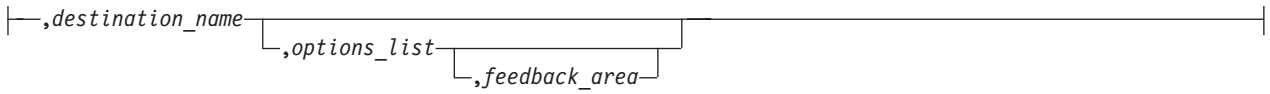
A:



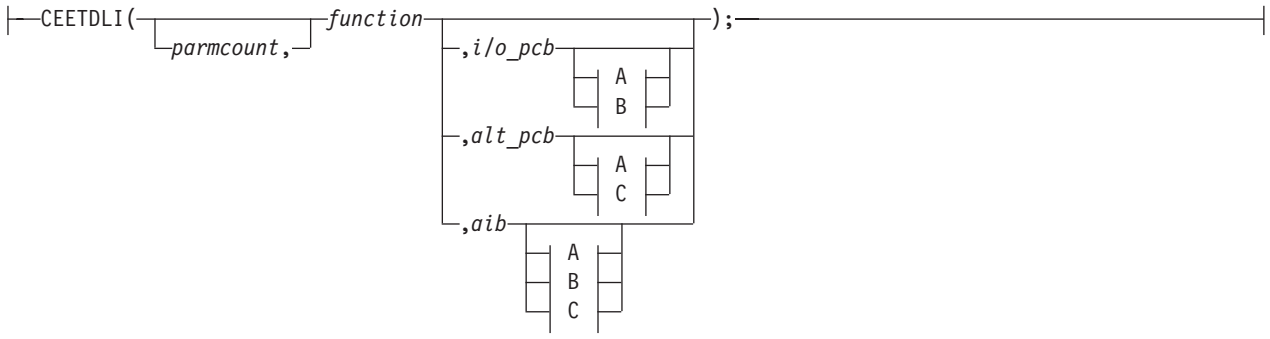
B:



C:



D:



Parameters

rr Receives the DL/I status or return code. It is a 2-character field shifted into the 2 lower bytes of an integer variable (*int*). If the status or return code is two blanks, 0 is placed in the field. You can test the rc parameter with an if statement; for example, `if (rc == 'IX')`. You can also use rc in a switch statement. You can choose to ignore the value placed in rc and use the status code returned in the program communication block (PCB) instead.

parmcount
Specifies the name of a fixed-binary (31) variable in user-defined storage that is a pointer to the number of parameters in the parameter list that follows *parmcount*. The *parmcount* field is a pointer to long.

function
Specifies the name of a character (4) variable, left-justified, in user-defined storage, which contains the call function to be used. The call function must be padded with blanks. For example, (GUbb is a call function.

i/o pcb
Specifies the address of the I/O PCB. The I/O PCB address is the first address passed on entry to the application program in the PCB list, given the following circumstances:

- A program executing in DLI or database management batch (DBB) regions where `CMPAT=YES` is coded on the PSB.
- Any program executing in batch message processing program (BMP), message processing program (MPP), or IMS Fast Path (IFP) regions regardless of the `CMPAT=` value.

alternate pcb
Specifies the name of a pointer variable that contains the address of the I/O PCB or alternate PCB to be used for the call. The PCB address must be one of the PCB addresses passed on entry to the application program in the PCB list.

aib
Specifies the name of the pointer variable that contains the address of the structure that defines the application interface block (AIB) in user-defined storage.

i/o area
Specifies the name of a pointer variable to a major structure, array, or character string that defines the I/O area in user-defined storage to be used for the call. The I/O area must be large enough to contain the returned data.

i/o area length
Specifies the name of a fixed-binary (31) variable in user-defined storage that contains the I/O area length.

area length
Specifies the name of a fixed-binary (31) variable in user-defined storage that contains the length of the area immediately following it in the parameter list. Up to seven area length/area pairs can be specified.

area
Specifies the name of the pointer variable that contains the address of the structure that defines the user-defined storage to be checkpointed. Up to seven area length/area pairs can be specified.

token
Specifies the name of a character (4) variable in user-defined storage that contains a user token.

options list
Specifies the name of the pointer variable that contains the address of the structure that defines the user-defined storage that contains processing options used with the call.

feedback area
Specifies the name of the pointer variable that contains the address of the structure that defines the user-defined storage that receives information about options list processing errors.

mod name
Specifies the name of a character (8) variable in user-defined storage that contains the user-defined MOD name used with the call. The *mod name* parameter is used only with MFS.

destination name
Specifies the name of a character (8) variable in user-defined storage that contains the name of the logical or terminal transaction code to which messages resulting from the call are sent.

I/O area

In C language, the I/O area can be of any type, including structure or array. The ceetdli declarations in leawi.h and the ctdli declarations in ims.h do not have any prototype information, so no type checking of the parameters is done. The I/O area can be auto, static, or allocated (with malloc or calloc). Give special consideration to C-strings because DL/I does not recognize the C convention of terminating strings with nulls ('\0'). Instead of using the strcpy and strcmp functions, you might want to use the memcpy and memcmp functions.

Example DL/I call formats

DL/I CEEDTLI interface:

```
#include <leawi.h>
ceetdli(function,aib,i/o_area)
```

DL/I AIBTDLI interface:

```
int rc;
⋮
rc = aibtdli(parmcount,function,aib,i/o_area)
```

DL/I language-specific interface:

```
#include <ims.h>
int rc;
⋮
rc = ctdli(function,i/o_pcb,i/o_area)
```

Related concepts:

"AIBTDLI interface" on page 245

Related reference:

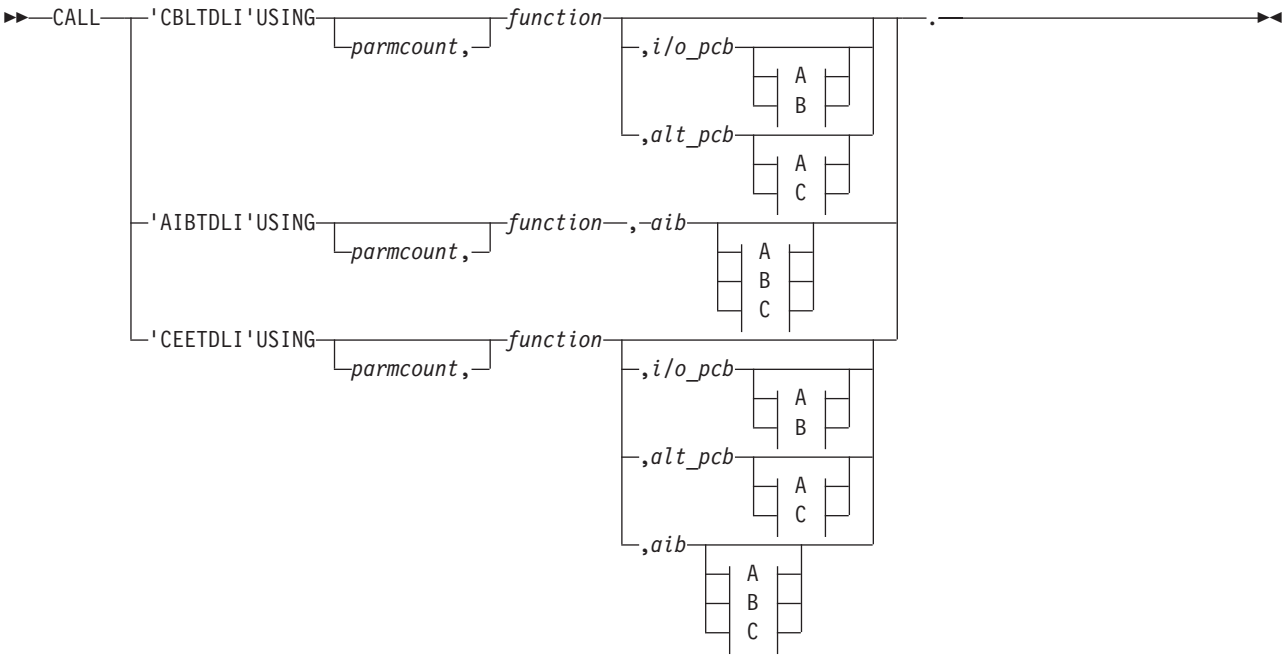
DL/I calls for transaction management (Application Programming APIs)

DL/I calls for IMS TM system services (Application Programming APIs)

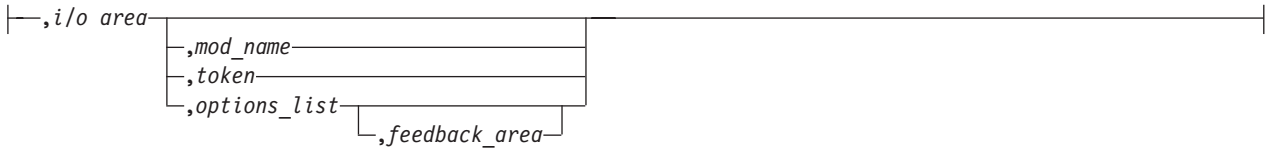
Application programming for COBOL

Application programs in COBOL use the following format, parameters, and DL/I calls to communicate with the IMS Transaction Manager.

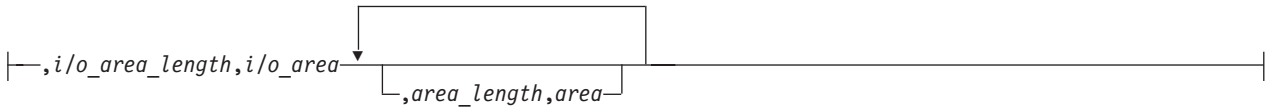
Format



A:



B:



C:

Parameters

parmcount
Specifies the identifier of a usage binary (4) byte data item in user-defined storage that contains the number of parameters in the parameter list that follows *parmcount*.

function
Specifies the identifier of a usage display (4) byte data item, left-justified, in

user-defined storage, which contains the call function to be used. The call function must be padded with blanks. For example, (GU⌂⌂ is a call function.

i/o pcb
Specifies the address of the I/O program communication block (PCB). The I/O PCB address is the first address passed on entry to the application program in the PCB list, given the following circumstances:

- A program executing in DLI or database management batch (DBB) regions where `CMPAT=YES` is coded on the PSB.
- Any program executing in batch message processing program (BMP), message processing program (MPP), or IMS Fast Path (IFP) regions regardless of the `CMPAT=` value.

alternate pcb
Specifies the identifier of the I/O PCB or alternate PCB group item from the PCB list that is passed to the application program on entry. This identifier is used for the call.

aib
Specifies the identifier of the group item that defines the application interface block (AIB) in user-defined storage.

i/o area
Specifies the identifier of a group item, table, or usage display data item that defines the I/O area to be used for the call. The I/O area must be large enough to contain the returned data.

i/o area length
Specifies the identifier of a usage binary (4) byte data item in user-defined storage that contains the I/O area length.

area length
Specifies the identifier of a usage binary (4) byte data item in user-defined storage that contains the length of the area immediately following it in the parameter list. Up to seven area length/area pairs can be specified.

area
Specifies the identifier of the group item that defines the area to be checkpointed. Up to seven area length/area pairs can be specified.

token
Specifies the identifier of a usage display (4) byte data item that contains a user token.

options list
Specifies the identifier of the group item that defines the user-defined storage that contains processing options used with the call.

feedback area
Specifies the identifier of the group item that defines the user-defined storage that receives information about options list processing errors.

mod name
Specifies the identifier of a usage display (8) byte data item in user-defined storage that contains the user-defined MOD name used with the call.

destination name
Specifies the identifier of a usage display (8) byte data item that contains the name of the logical terminal or transaction code to which messages resulting from the call are sent.

Example DL/I call formats

DL/I CEETDLI interface:

```
CALL 'CEETDLI' USING function, aib,i/o area.
```

DL/I AIBTDLI interface:

```
CALL 'AIBTDLI' USING function, aib,i/o area.
```

DL/I language-specific interface:

```
CALL 'CBLTDLI' USING function, i/o pcb, i/o area.
```

Related concepts:

"AIBTDLI interface" on page 245

Related reference:

DL/I calls for transaction management (Application Programming APIs)

DL/I calls for IMS TM system services (Application Programming APIs)

Java application programming for IMS

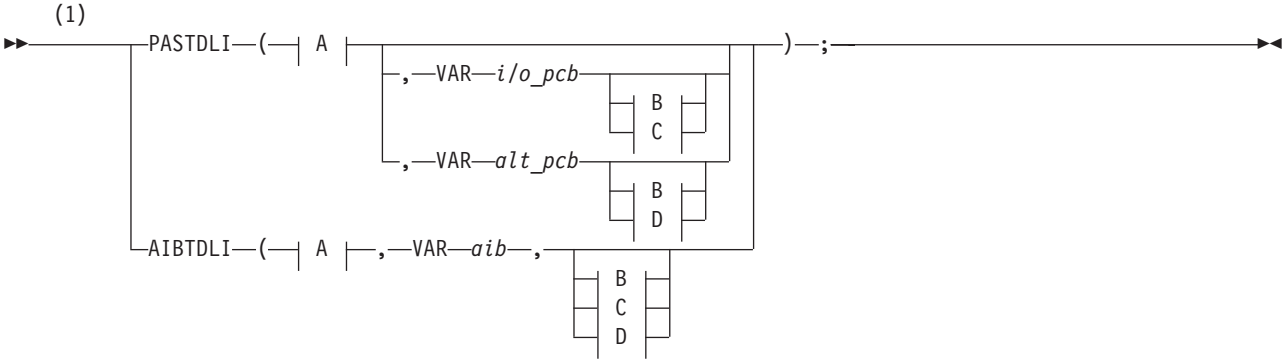
IMS provides support for developing applications using the Java programming language.

You can write Java applications to access IMS databases and process IMS transactions by using the drivers and resource adapters of the IMS solutions for Java development.

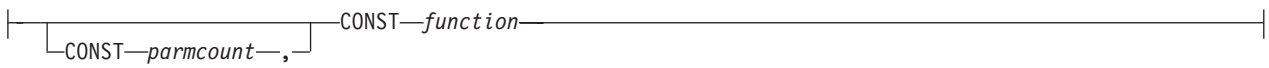
Application programming for Pascal

Application programs in Pascal use the following format, parameters, and DL/I calls to communicate with the IMS Transaction Manager.

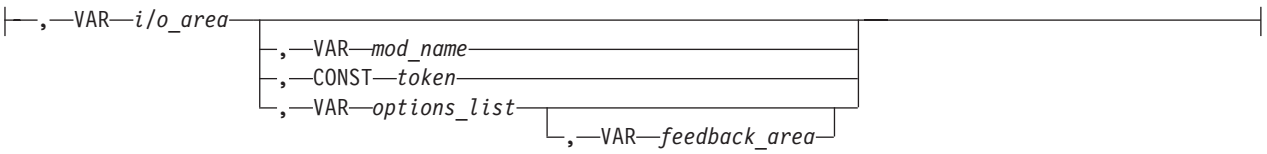
Format



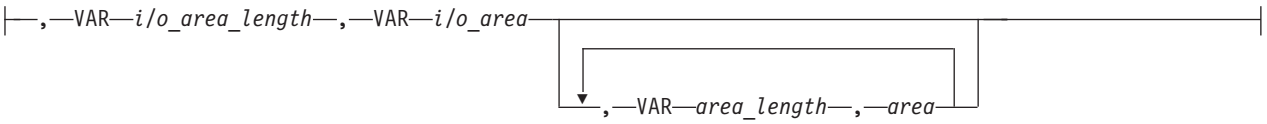
A:



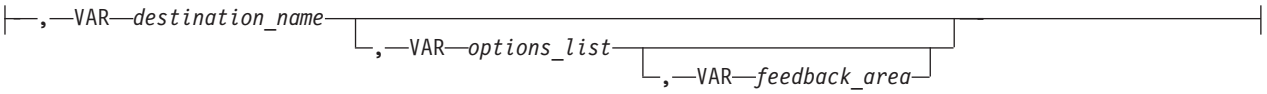
B:



C:



D:



Notes:

1 For AIBTDLI, *parmcount* is required for applications.

Parameters

parmcount
specifies the address of a fixed-binary (31) variable in user-defined storage that contains the number of parameters in the parameter list that follows *parmcount*.

function
Specifies the name of a character (4) variable, left-justified, in user-defined storage, which contains the call function to be used. The call function must be padded with blanks. For example, (GU△△ is a call function.

i/o pcb
Specifies the address of the program communication block (I/O PCB). The I/O PCB address is the first address passed on entry to the application program in the PCB list, given the following circumstances:

- A program executing in DLI or database management batch (DBB) regions where `CMPAT=YES` is coded on the PSB.
- Any program executing in batch message processing program (BMP), message processing program (MPP), or IMS Fast Path (IFP) regions regardless of the `CMPAT=` value.

alternate pcb
Specifies the name of a pointer variable that contains the address of the I/O PCB defined in the call procedure statement.

aib
Specifies the name of a pointer variable that contains the address of the structure that defines the application interface block (AIB) in user-defined storage.

i/o area
Specifies the name of a pointer variable to a major structure, array, or character

string that defines the I/O area in user-defined storage to be used for the call. The I/O area must be large enough to contain the returned data.

i/o area length
Specifies the name of a fixed-binary (31) variable in user-defined storage that contains the I/O area length.

area length
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the length (specified in binary) of the area immediately following it in the parameter list. Up to seven area length/area pairs can be specified.

area
Specifies the name of a pointer variable that contains the address of the structure that defines the area in user-defined storage to be checkpointed. Up to seven area length/area pairs can be specified.

token
Specifies the name of a character (4) variable in user-defined storage that contains a user token.

options list
Specifies the name of a pointer variable that contains the address of the structure that defines the user-defined storage that contains processing options used with the call.

feedback area
Specifies the name of the pointer variable that contains the address of the structure that defines the user-defined storage that receives information about options list processing errors.

mod name
Specifies the name of a character (8) variable in user-defined storage that contains the user-defined MOD name used with the call.

destination name
Specifies the name of a character (8) variable in user-defined storage that contains the name of the logical terminal or transaction code to which messages resulting from the call are sent.

Example DL/I call formats

DL/I AIBTDLI interface:

```
AIBTDLI(CONST function,
        VAR aib,
        VAR I/O area);
```

DL/I language-specific interface:

```
PASTDLI(CONST function,
area    VAR I/O PCB
        VAR I/O area);
```

Related concepts:

"AIBTDLI interface" on page 245

Related reference:

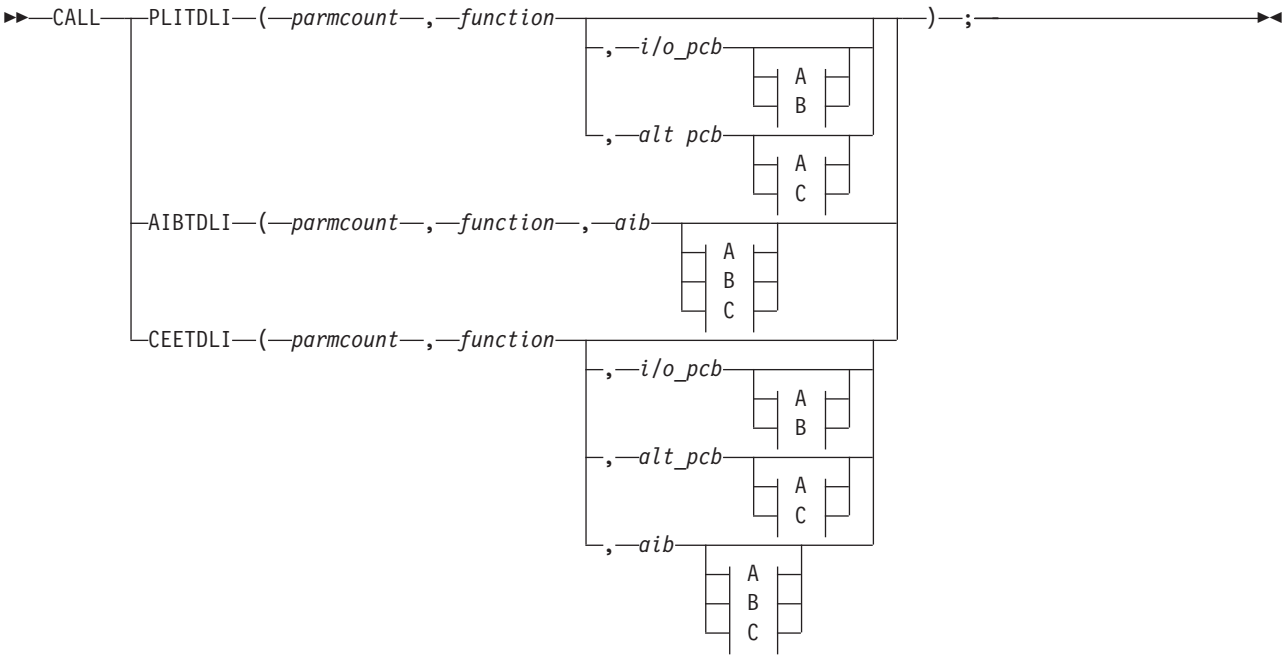
DL/I calls for transaction management (Application Programming APIs)

DL/I calls for IMS TM system services (Application Programming APIs)

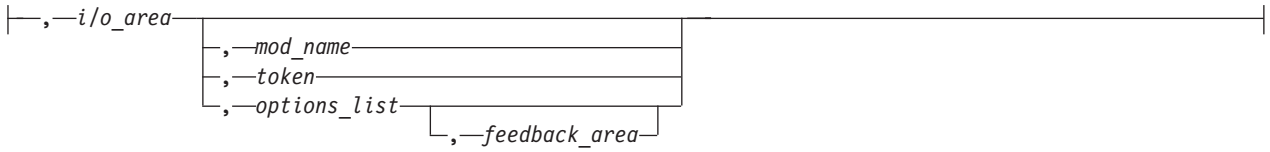
Application programming for PL/I

Application programs in PL/I use the following format, parameters, and DL/I calls to communicate with the IMS Transaction Manager.

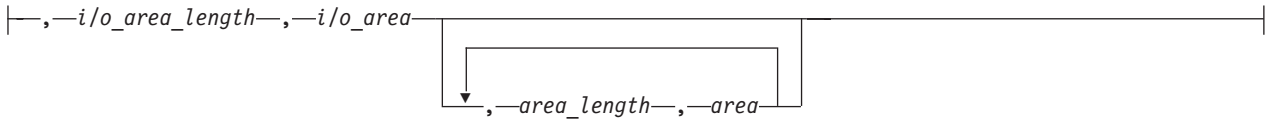
Format



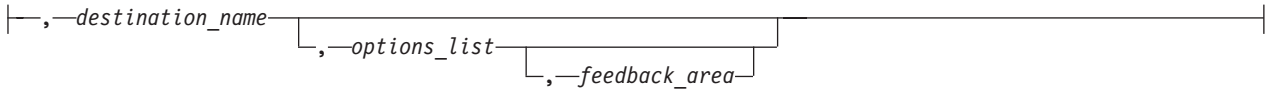
A:



B:



C:



Parameters

parmcount
Specifies the name of a fixed-binary (31-byte) variable that contains the number of arguments that follow *parmcount*.

function
Specifies the name of a character (4-byte) variable, left justified, blank padded character string that contains the call function to be used. For example, (GU␣␣ is a call function.

i/o pcb
Specifies the address of the program communication block (I/O PCB). The I/O PCB address is the first address passed on entry to the application program in the PCB list, given the following circumstances:

- A program executing in DLI or DBB regions where `CMPAT=YES` is coded on the PSB.
- Any program executing in batch message processing program (BMP), message processing program (MPP), or IMS Fast Path (IFP) regions regardless of the `CMPAT=` value.

alternate pcb
Specifies the structure associated with the I/O PCB or alternate PCB to be used for the call. This structure is based on a PCB address that must be one of the PCB addresses passed on entry to the application program.

aib
Specifies the name of the structure that defines the application interface block (AIB).

i/o area
Specifies the name of the I/O area used for the call. The I/O area must be large enough to contain the returned data.

i/o area length
Specifies the name of a fixed binary (31) variable in user-defined storage that contains the I/O area length (specified in binary).

area length
Specifies the name of a fixed binary (31) variable that contains the length (specified in binary) of the area immediately following it in the parameter list. Up to seven area length/area pairs can be specified.

area
Specifies the name of the area to be checkpointed. Up to seven area length/area pairs can be specified.

token
Specifies the name of a character (4) variable that contains a user token.

options list
Specifies the name of a structure that contains processing options used with the call.

feedback area
Specifies the name of a structure that receives information about options list processing errors.

mod name
Specifies the name of a character (8) variable character string containing the user-defined MOD name used with the call.

destination name
Specifies the name of a character (8) variable character string containing the logical terminal or transaction code to which messages resulting from the call are sent.

Example DL/I call formats

DL/I CEETDLI interface:

```
%INCLUDE CEEIBMAW;
CALL CEETDLI (function, i/o pcb, i/o area);
```

DL/I AIBTDLI interface:

```
CALL AIBTDLI (parmcount, function, aib, i/o area);
```

DL/I language-specific interface:

```
CALL PLITDLI (parmcount, function, i/o pcb, i/o area);
```

Relationship of calls to PCB types

The following table shows the relationship of DL/I calls to I/O and alternate program communication blocks (PCBs).

The PCB can be specified as a parameter in the call list, or in the AIB, depending on which xxxTDLI interface is used:

Table 65. Call relationship to PCBs and AIBs.

Call	I/O PCBs	ALT PCBs
APSB [1]		
AUTH	X	
CHKP (basic)	X	
CHKP (symbolic)	X	
CHNG [2]		X
CMD	X	
DPSB [1]		
GCMD	X	
GN	X	
GSCD	X	
GU	X	
INIT	X	
INQY	X	X
ISRT	X	X
LOG	X	
PURG	X	X
ROLB	X	
ROLS	X	
ROLL [1]		
SETO	X	X
SETS	X	

Table 65. Call relationship to PCBs and AIBs (continued).

Call	I/O PCBs	ALT PCBs
SETU	X	
SYNC	X	
XRST	X	

Notes:

1. This call is not associated with a PCB.
2. The alternate PCB used by this call must be modifiable.

Specifying the I/O PCB mask

After your program issues a call with the I/O program communications block (PCB), IMS returns information about the results of the call to the I/O PCB. To determine the results of the call, your program must check the information that IMS returns.

Issuing a system service call requires an I/O PCB. Because the I/O PCB resides outside your program, you must define a mask of the PCB in your program to check the results of IMS calls. The mask must contain the same fields, in the same order, as the I/O PCB. Your program can then refer to the fields in the PCB through the PCB mask.

An I/O PCB contains the fields listed in the following table. The table describes these fields, their lengths, and which environments are applicable for each field.

Table 66. I/O PCB mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Logical terminal name [1]	8	X		X		
Reserved for IMS [2]	2	X		X		
Status code [3]	2	X	X	X	X	X
4-byte Local date and time [4]						
Date	2	X		X		
Time	2	X		X		
Input message sequence number [5]	4	X		X		
Message output descriptor name [6]	8	X		X		
Userid [7]	8	X		X		
Group name [8]	8	X		X		
12-Byte Time Stamp [9]						
Date	4	X		X		
Time	6	X		X		
UTC Offset	2	X		X		
Userid Indicator[10]	1	X		X		

Table 66. I/O PCB mask (continued)

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Reserved for IMS[2]	3					

Note:

1. **Logical Terminal Name**

   This field contains the name of the terminal that sent the message. When your program retrieves an input message, IMS places the name of the logical terminal that sent the message in this field. When you want to send a message back to this terminal, you refer to the I/O PCB when you issue the `ISRT` call, and IMS takes the name of the logical terminal from the I/O PCB as the destination.

2. **Reserved for IMS**

   These fields are reserved.

3. **Status Code**

   IMS places the status code describing the result of the DL/I call in this field. IMS updates the status code after each DL/I call that the program issues. Your program should always test the status code after issuing a DL/I call.

   The three status code categories are:

   - Successful status codes or status codes with exceptional but valid conditions. This category does not contain errors. If the call was completely successful, this field contains blanks. Many of the codes in this category are for information only. For example, a QC status code means that no more messages exist in the message queue for the program. When your program receives this status code, it should terminate.
   - Programming errors. The errors in this category are usually ones that you can correct. For example, an AD status code indicates an invalid function code.
   - I/O or system errors.

   For the second and third categories, your program should have an error routine that prints information about the last call that was issued program termination. Most installations have a standard error routine that all application programs at the installation use.

4. **Local Date and Time**

   The current local date and time are in the prefix of all input messages except those originating from non-message-driven BMPs. The local date is a packed-decimal, right-aligned date, in the format `yyddd`. The local time is a packed-decimal time in the format `hhmmsst`. The current local date and time indicate when IMS received the entire message and enqueued it as input for the program, rather than the time that the application program received the message. To obtain the application processing time, you must use the time facility of the programming language you are using.

   For a conversation, for an input message originating from a program or for a message received using Multiple System Coupling (MSC), the time and date indicate when the original message was received from the terminal.

   Note: Be careful when comparing the local date and time in the I/O PCB with the current time returned by the operating system. The I/O PCB date and time may not be consistent with the current time. It may even be greater than the current time for the following reasons:

- The time stamp in the I/O PCB is the local time that the message was received by IMS. If the local time was changed after the message arrived, it is possible for the current time to appear to be earlier than the I/O PCB time. This effect would be likely to occur in the hour immediately after the fall time change, when the clock is set back by one hour.
- The time stamp in the I/O PCB is derived from an internal IMS time stamp stored with the message. This internal time stamp is in Coordinated Universal Time (UTC), and contains the time zone offset that was in effect at the time the message was enqueued. This time zone offset is added to the UTC time to obtain the local time that is placed in the I/O PCB. However, the time zone offset that is stored is only fifteen minutes. If the real time zone offset was not an integer multiple of fifteen minutes, the local time passed back in the I/O PCB will differ from the actual time by plus or minus 7.5 minutes. This could cause the I/O PCB time to be later than the current time. See *IMS Version 11 Operations and Automation* for further explanation.

Concerns about the value in the local time stamp in the I/O PCB can be reduced by using the extended time stamp introduced in IMS V6. The system administrator can choose the format of the extended time stamp to be either local time or UTC. In some situations, it may be advantageous for the application to request the time in UTC from the operating system and compare it to the UTC form of the extended time stamp. This is an option available in installations where there is no ETR to keep the IMS UTC offset in sync with the z/OS UTC offset over changes in local time.

5. **Input Message Sequence Number**

   The input message sequence number is in the prefix of all input messages except those originating from non-message-driven BMPs. This field contains the sequence number IMS assigned to the input message. The number is binary. IMS assigns sequence numbers by physical terminal, which are continuous since the time of the most recent IMS startup.

6. **Message Output Descriptor Name**

   You only use this field when you use MFS. When you issue a `GU` call with a message output descriptor (MOD), IMS places its name in this area. If your program encounters an error, it can change the format of the screen and send an error message to the terminal by using this field. To do this, the program must change the MOD name by including the MOD name parameter on an `ISRT` or `PURG` call.

   Although MFS does not support APPC, LU 6.2 programs can use an interface to emulate MFS. For example, the application program can use the MOD name to communicate with IMS to specify how an error message is to be formatted.

   Related reading: For more information on the MOD name and the LTERM interface, see *IMS Version 11 Communications and Connections*.

7. **Userid**

   The use of this field is connected with RACF signon security. If signon is not active in the system, this field contains blanks.

   If signon is active in the system, the field contains one of the following:

   - The user's identification from the source terminal.
   - The LTERM name of the source terminal if signon is not active for that terminal.

- The authorization ID. For batch-oriented BMPs, the authorization ID is dependent on the value specified for the BMPUSID= keyword in the DFSDCxxx PROCLIB member:
  - If BMPUSID=USERID is specified, the value from the USER= keyword on the JOB statement is used.
  - If USER= is not specified on the JOB statement, the program's PSB name is used.
  - If BMPUSID=PSBNAME is specified, or if BMPUSID= is not specified at all, the program's PSB name is used.

    Related Reading: For more information about authorizing resource use in a dependent region, see *IMS Version 11 System Administration*.

8. **Group Name**

   The group name, which is used by DB2 to provide security for SQL calls, is created through IMS transactions.

   Three instances that apply to the group name are:

   - If you use RACF and signon on your IMS system, the RACROUTE SAF (extract) call returns an eight-character group name.
   - If you use your own security package on your IMS system, the RACROUTE SAF call returns any eight-character name from the package and treats it as a group name. If the RACROUTE SAF call returns a return code of 4 or 8, a group name was not returned, and IMS blanks out the group name field.
   - If you use LU 6.2, the transaction header can contain a group name.

     Related reading: See *IMS Version 11 Communications and Connections* for more information on LU 6.2.

9. **12-Byte Time Stamp**

   This field contains the current date and time fields, but in the IMS internal packed-decimal format. The time stamp has the following parts:

   Date `yyyydddf`

   This packed-decimal date contains the year (`yyyy`), day of the year (`ddd`), and a valid packed-decimal + sign such as (`f`).

   Time `hhmmssthmiju`

   This packed-decimal time consists of hours, minutes, and seconds (`hhmmss`) and fractions of the second to the microsecond (`thmiju`). No packed-decimal sign is affixed to this part of the time stamp.

   UTC Offset
   `aqq$`

   The packed-decimal UTC offset is prefixed by 4 bits of attributes (`a`). If the 4th bit of (`a`) is 0, the time stamp is UTC; otherwise, the time stamp is local time. The control region parameter, `TSR=(U/L)`, specified in the DFSPBxxx PROCLIB member, controls the representation of the time stamp with respect to local time versus UTC time.

   The offset value (`qq$`) is the number of quarter hours of offset to be added to UTC or local time to convert to local or UTC time respectively.

   The offset sign (`$`) follows the convention for a packed-decimal plus or minus sign.

   Field 4 on the I/O PCB Mask always contains the local date and time. For a description of field 4, see the notes for the previous table.

Related reading: For a more detailed description of the internal packed-decimal time-format, see *IMS Version 11 System Utilities*.

10. **Userid Indicator**

    The Userid Indicator is provided in the I/O PCB and in the response to the INQY call. The Userid Indicator contains one of the following:

    - U - The user's identification from the source terminal during signon
    - L - The LTERM name of the source terminal if signon is not active
    - P - The PSBNAME of the source BMP or transaction
    - O - Other name

    The value contained in the Userid Indicator field indicates the contents of the userid field.

Related concepts:

"Results of a message: I/O PCB" on page 398

Specifying the alternate PCB mask

An alternate program communication block (PCB) mask contains three fields.

The following table describes these fields, the field length, and in which environment the field applies.

Table 67. Alternate PCB mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Logical terminal name [1]	8 bytes	X		X		
Reserved for IMS [2]	2 bytes	X		X		
Status code [3]	2 bytes	X		X		

Note:

1. **Logical Terminal Name**

   This field contains the name of the logical terminal, LU 6.2 descriptor or the transaction code to which you want to send the message.

   Related reading: For more information on LU 6.2, see *IMS Version 11 Communications and Connections*.

2. **Reserved for IMS**

   This 2-byte field is reserved.

3. **Status Code**

   This field contains the 2-byte status code that describes the results of the call that used this PCB most recently.

Related concepts:

"Sending messages to other terminals and programs" on page 408

Specifying the AIB mask

The AIB is used by your program to communicate with IMS, when your application does not have a program communication block (PCB) address or the call function does not use a PCB.

The application program can use the returned PCB address, when available, to inspect the status code in the PCB and to obtain any other information needed by

the application program. The AIB mask enables your program to interpret the control block defined. The AIB structure must be defined in working storage, on a fullword boundary, and initialized according to the order and byte length of the fields as shown in the following table. The table's notes describe the contents of each field.

Table 68. AIB fields

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
AIB identifier [1]	8	X	X	X	X	X
DFSAIB allocated length [2]	4	X	X	X	X	X
Subfunction code [3]	8	X	X	X	X	X
Resource name 1[4]	8	X	X	X	X	X
Reserved 1 [5]	8					
Resource name 2 [6]	8					
Maximum output area length [7]	4	X	X	X	X	X
Output area length used [8]	4	X	X	X	X	X
AIBRSFLD [9]	4					
Reserved 2 [10]	8					
Return code [11]	4	X	X	X	X	X
Reason code [12]	4	X	X	X	X	X
Error code extension [13]	4	X		X		
Resource address [14]	4	X	X	X	X	X
Reserved 3 [15]	40					

Note:

1. **AIB Identifier (AIBID)**

   This 8-byte field contains the AIB identifier. You must initialize AIBID in your application program to the value DFSAIBbb before you issue DL/I calls. This field is required. When the call is completed, the information returned in this field is unchanged.

2. **DFSAIB Allocated Length (AIBLEN)**

   This field contains the actual 4-byte length of the AIB as defined by your program. You must initialize AIBLEN in your application program before you issue DL/I calls. The minimum length required is 128 bytes. When the call is completed, the information returned in this field is unchanged. This field is required.

3. **Subfunction Code (AIBSFUNC)**

   This 8-byte field contains the subfunction code for those calls that use a subfunction. You must initialize AIBSFUNC in your application program before you issue DL/I calls. When the call is completed, the information returned in this field is unchanged.

4. **Resource Name (AIBRSNM1)**

This 8-byte field contains the name of a resource. The resource varies depending on the call. You must initialize AIBRSNM1 in your application program before you issue DL/I calls. When the call is complete, the information returned in this field is unchanged. This field is required.

For PCB related calls where the AIB is used to pass the PCB name instead of passing the PCB address in the call list, this field contains the PCB name. The PCB name for the I/O PCB is IOPCBbb The PCB name for other types of PCBs is defined in the PCBNAME= parameter in PSBGEN.

5. **Reserved 1**

   This 16-byte field is reserved.

6. **Resource name 2**

   This 8-byte field is reserved.

7. **Maximum Output Area Length (AIBOALEN)**

   This 4-byte field contains the length of the output area in bytes that was specified in the call list. You must initialize AIBOALEN in your application program for all calls that return data to the output area. When the call is completed, the information returned in this area is unchanged.

8. **Used Output Area Length (AIBOAUSE)**

   This 4-byte field contains the length of the data returned by IMS for all calls that return data to the output area. When the call is completed this field contains the length of the I/O area used for this call.

9. **Reserved 2**

   This 8-byte field is reserved.

   The first four bytes are used by the ICAL call to specify the time to wait for the synchronous call process to complete (AIBRSFLD).

10. **Return code (AIBRETRN)**

    When the call is completed, this 4-byte field contains the return code.

11. **AIBRSFLD**

    This 4-byte field contains a resource information. The usage of this field varies depending on the call.

12. **Reason Code (AIBREASN)**

    When the call is completed, this 4-byte field contains the reason code.

13. **Error Code Extension (AIBERRXT)**

    This 4-byte field contains additional error information depending on the return code in AIBRETRN and the reason code in AIBREASN.

14. **Resource Address (AIBRSA1)**

    When the call is completed, this 4-byte field contains call-specific information. For PCB related calls where the AIB is used to pass the PCB name instead of passing the PCB address in the call list, this field returns the PCB address.

15. **Reserved 3**

    This 40-byte field is reserved.

Specifying the I/O areas

Use an I/O area to pass segments between the application program and IMS.

What the I/O area contains depends on the type of call you are issuing:

- When you retrieve a segment, IMS places the segment you requested in the I/O area.

- When you add a new segment, you first build the new segment in the I/O area.
- Before modifying a segment, your program must first retrieve it. When you retrieve the segment, IMS places the segment in an I/O area.

The format of the record segments you pass between your program and IMS can be fixed length or variable length. Only one difference is important to the application program: a message segment containing a 2-byte length field (or 4 bytes for the PLITDLI interface) at the beginning of the data area of the segment.

The I/O area for IMS calls must be large enough to hold the largest segment your program retrieves from or sends to IMS.

If your program issues any Get or ISRT calls that use the D command code, the I/O area must be large enough to hold the largest path of segments that the program retrieves or inserts.

AIBTDLI interface

Use AIBTDLI as the interface between your application program and IMS.

Restriction: No fields in the AIB can be used by the application program except as defined by IMS.

When you use the AIBTDLI interface, you specify the program communication block (PCB) requested for the call by placing the PCB name (as defined by PSBGEN) in the resource name field of the AIB. You do not specify the PCB address. Because the AIB contains the PCB name, your application program can refer to the PCB name rather than the PCB address. Your application program does not need to know the relative PCB position in the PCB list. At completion of the call, the AIB returns the PCB address that corresponds to the PCB name passed by the application program.

The names of DB PCBs and alternate PCBs are defined by the user during PSBGEN. All I/O PCBs are generated with the PCB name ␣␣␣ For a generated program specification block (GPSB), the I/O PCB is generated with the PCB name IOPCB␣␣␣, and the modifiable alternate PCB is generated with the PCB name TPPCB1␣␣.

The ability to pass the PCB name means that you do not need to know the relative PCB number in the PCB list. In addition, the AIBTDLI interface enables your application program to make calls on PCBs that do not reside in the PCB list. The LIST= keyword, which is defined in the PCB macro during PSBGEN, controls whether the PCB is included in the PCB list.

The AIB resides in user-defined storage that is passed to IMS for DL/I calls that use the AIBTDLI interface. Upon call completion, IMS updates the AIB. Allocate at least 128 bytes of storage for the AIB.

Related concepts:

"PCB masks for GSAM databases" on page 305

Related reference:

"Application programming for PL/I" on page 376

"Application programming for Pascal" on page 373

"Application programming for C language" on page 368

"Application programming for assembler language" on page 365

"Assembler language application programming" on page 213

Specifying language-specific entry points

IMS gives control to an application program through an entry point. Use the correct format for coding entry statements in assembler language, C language, COBOL, Pascal, and PL/I.

Your entry point must refer to the program communication blocks (PCBs) in the order in which they are defined in the PSB.

IMS passes the PCB pointers to a PL/I program differently than it passes them to an assembler language, C language, COBOL, Java, or Pascal program. In addition, Pascal requires that IMS pass an integer before passing the PCB pointers. IMS uses the LANG keyword or the PSBGEN statement of PSBGEN to determine the type of program to which it is passing control. Therefore, you must be sure that the language specified during PSBGEN is consistent with the language of the program.

Application interfaces that use the AIB structure (AIBTDLI or CEETDLI) use the PCB name rather than the PCB structure and do not require the PCB list to be passed at entry to the application program.

When you code each DL/I call, you must provide the PCB you want to use for that call. For all IMS TM application programs, the list of PCBs the program can access is passed to the program at its entry point.

Assembler language

You can use any name for the entry statement to an assembler language DL/I program. When IMS passes control to the application program, register 1 contains the address of a variable-length fullword parameter list. Each word in the list contains the address of a PCB. Save the parameter list address before you overwrite the contents of register 1. IMS sets the high-order byte of the last fullword in the list to X'80' to indicate the end of the list. Use standard z/OS linkage conventions with forward and backward chaining.

C language

When IMS passes control to your program, it passes the addresses, in the form of pointers, for each of the PCBs your program uses. The usual argc and argv arguments are not available to a program invoked by IMS. The IMS parameter list is made accessible by using the __pcblist macro. You can directly reference the PCBs by __pcblist[0], __pcblist[1], or you can define macros to give these more meaningful names. I/O PCBs must be cast to get the proper type:

```
(IO_PCB_TYPE *)(__pcblist[0])
```

The entry statement for a C language program is the main statement.

```
#pragma runopts(env(IMS),plist(IMS))
#include <ims.h>

main()
{
.
.
.
}
```

The env option specifies the operating environment in which your C language program is to run. For example, if your C language program is invoked under IMS and uses IMS facilities, specify env(IMS). The plist option specifies the format of the invocation parameters received by your C language program when it is invoked. When your program is invoked by a system support services program such as IMS, the format of the parameters passed to your main program must be converted into the C language format: argv, argc, and envp. To do this conversion, you must specify the format of the parameter list received by your C language program. The ims.h include file contains declarations for PCB masks.

You can finish program execution in three ways:

- End the main procedure without an explicit return statement.
- Execute a return statement from main.
- Execute an exit or an abort call from anywhere, or alternately issue a longjmp back to main, and then do a normal return.

One C language program can pass control to another by using the system function. The normal rules for passing parameters apply. For example, when using the system function, the argc and argv arguments can be used to pass information. The initial __pcblist is made available to the invoked program.

COBOL

The procedure statement must refer to the I/O PCB first, then to any alternate PCB it uses, and finally to the DB PCBs it uses. The alternate PCBs and DB PCBs must be listed in the order in which they are defined in the PSB.

```
Procedure division using the PCB-NAME-1 [,...,PCB-NAME-N]
```

On previous versions of IMS, the `using` keyword might be coded on the entry statement to reference PCBs. However, IMS continues to accept such coding on the entry statement.

Recommendation: Use the procedure statement rather than the entry statement to reference the PCBs.

Pascal

The entry point must be declared as a REENTRANT procedure. When IMS passes control to a Pascal procedure, the first address in the parameter list is reserved for Pascal's use and the other addresses are the PCBs the program uses. The PCB types must be defined before this entry statement. The IMS interface routine PASTDLI must be declared with the GENERIC directive.

```
procedure ANYNAME(var SAVE: INTEGER;
                  var pcb1-name: pcb1-name-type[;
                  ...
                  var pcbn-name: pcbn-name-type]); REENTRANT;
procedure ANYNAME;
(* Any local declarations *)
  procedure PASTDLI; GENERIC;
begin
  (* Code for ANYNAME *)
end;
```

PL/I

The entry statement can be any valid PL/I name and must appear as the first executable statement in the program. When IMS passes control to your program, it passes the addresses of each of the PCBs your program uses in the form of pointers. When you code the entry statement, make sure you code the parameters of this statement as pointers to the PCBs, and not the PCB names.

```
anyname: PROCEDURE (pcb1_ptr [,..., pcbn_ptr]) OPTIONS (MAIN);
⋮
RETURN;
```

CCETDLI and AIBTDLI interface considerations

The CCETDLI considerations are:

- For PL/I programs, the CEETDLI entry point is defined in the CEEIBMAW include file. Alternatively, you can declare it yourself. But it must be declared as an assembler language entry (DCL CEETDLI OPTIONS(ASM);).
- For C language applications, you must specify env(IMS) and plist(IMS); these specifications enable the application to accept the PCB list of arguments. The CEETDLI function is defined in <leawi.h>; the CTDLI function is defined in <ims.h>.

The AIBTDLI considerations are:

- When using the AIBTDLI interface for C/MVS, COBOL, or PL/I language applications, the language run-time options for suppressing abend interception (that is, NOSPIE and NOSTAE) must be specified. However, for Language Environment-conforming applications, the NOSPIE and NOSTAE restriction is removed.
- The AIBTDLI entry point for PL/I programs must be declared as an assembler language entry (DCL AIBTDLI OPTIONS(ASM);).
- For C language applications, you must specify env(IMS) and plist(IMS); these specifications enable the application to accept the PCB list of arguments.

Program communication block (PCB) lists

Use the correct format of program communication block (PCB) lists and generated program specification block (GPSB) PCB lists in your application program.

PCB list format

This is the format of a PCB:

```
[IOPCB]
[Alternate PCB ... Alternate PCB]
[DB PCB ... DB PCB]
[GSAM PCB ... GSAM PCB]
```

Each PSB must contain at least one PCB. An I/O PCB or alternate PCB is required for transaction management calls, and an I/O PCB is required for most system service calls. DB PCBs for DL/I databases are used only with the IMS Database Manager, but can be present even though your program is running under DCCTL or TM Batch. (A DB PCB can be a full-function PCB, a DEDB PCB, or an MSDB PCB.) GSAM PCBs can be used with DCCTL or TM batch.

Format of a GPSB PCB list

A generated program specification block (GPSB) has the following format:

```
[IOPCB]
[Alternate PCB]
```

A GPSB contains only an I/O PCB and one modifiable alternate PCB. It can be used by all transaction management application programs, and permits access to the PCBs specified without the need for PSBGEN.

The PCBs in a GPSB have predefined PCB names. The name of the I/O PCB is IOPCBbbb. The name of the alternate PCB is TPPCB1bbb.

PCB summary

I/O PCBs and alternate PCBs can be used in various types of application programs.

TM Batch Programs
Alternate PCBs are always included in the list of PCBs supplied to the program by IMS TM. The I/O PCB is always present in the PCB list regardless of the CMPAT options specified in PSBGEN.

BMPs, MPPs, and IFPs
The I/O PCB is always present in the PCB list and is always the first address in the list, regardless of the CMPAT options specified in the PSB. The PCB list always contains the address of the I/O PCB followed by the addresses of any alternate PCBs, followed by the addresses of the DB PCBs.

Language environments

IBM Language Environment provides the strategic execution environment for running your application programs written in one or more high level languages.

It provides not only language-specific run-time support, but also cross-language run-time services for your applications, such as support for initialization, termination, message handling, condition handling, storage management, and National Language Support. Many of Language Environment's services are accessible explicitly through a set of Language Environment interfaces that are common across programming languages; these services are accessible from any Language Environment-conforming program.

Language Environment-conforming programs can be compiled with the following compilers:

- IBM C++/MVS
- IBM COBOL
- IBM PL/I

The CEETDLI interface to IMS

The language-independent CEETDLI interface to IMS is provided by Language Environment. It is the only IMS interface that supports the advanced error handling capabilities provided by Language Environment. The CEETDLI interface supports the same functionality as the other IMS application interfaces, and it has the following characteristics:

- The parmcount variable is optional.
- Length fields are 2 bytes long.
- Direct pointers are used.

Related reading: For more information about Language Environment, see *z/OS Language Environment Programming Guide*.

LANG= option on PSBGEN for PL/I compatibility

For IMS PL/I applications running in a compatibility mode that uses the PLICALLA entry point, you must specify LANG=PLI on the PSBGEN. Your other option is to change the entry point and add SYSTEM(IMS) to the EXEC PARM of the compile step so that you can specify LANG=blank or LANG=PLI on the PSBGEN. The following table summarizes when you can use LANG=blank and LANG=PLI.

Table 69. Using LANG= option in a Language Environment for PL/I compatibility

Compile exec statement is PARM=(...,SYSTEM(IMS)...	Entry point name is PLICALLA	Valid LANG= value
Yes	Yes	LANG=PLI
Yes	No	LANG=blank or LANG=PLI
No	No	**Note:** Not valid for IMS PL/I applications
No	Yes	LANG=PLI

PLICALLA is only valid for PL/I compatibility with Language Environment. If a PL/I application using PLICALLA entry at bind time is bound using Language Environment with the PLICALLA entry, the bind will work; however, you must specify LANG=PLI in the PSB. If the application is re-compiled using PL/I for z/OS & VM Version 1 Release 1 or later, and then bound using Language Environment Version 1 Release 2 or later, the bind will fail. You must remove the PLICALLA entry statement from the bind.

Special DL/I situations for IMS TM programming

Special considerations during application programming for IMS Transaction Manager include mixed-language programming, using the extended addressing capabilities of z/OS, COBOL compiler options for preloaded programs, and considerations for the DCCTL environment.

Mixed-language programming

When an application program uses the Language Environment language-independent interface, CEETDLI, IMS does not need to know the language of the calling program.

When the application program calls IMS in a language-dependent interface, IMS determines the language of the calling program according to the entry name specified in the CALL statement:

- CALL CBLTDLI indicates the program is in COBOL.
- CALL PLITDLI indicates the program is in PL/I.
- CALL PASTDLI indicates the program is in Pascal.

- ctdli(...) indicates the program is in C language.
- CALL ASMTDLI indicates the program is in assembler language.

If a PL/I program calls an assembler language subroutine and the assembler language subroutine makes DL/I calls by using CALL ASMTDLI, the assembler language subroutine should use the assembler language calling convention, not the PL/I convention.

In this situation, where the I/O area uses the LLZZ format, the LL is a halfword, not the fullword that is used for PLITDLI.

Using Language Environment routine retention

If you run programs in an IMS TM dependent region that requires Language Environment (such as an IMS message processing region), you can improve performance if you use Language Environment library routine retention along with the existing PREINIT feature of IMS TM.

Related reading: For more information about Language Environment, see *z/OS Language Environment Programming Guide*.

Using the extended addressing capabilities of z/OS

The two modes inz/OS with extended addressing capabilities are: the addressing mode (AMODE) and the residency mode (RMODE).

IMS places no constraints on the RMODE and AMODE of an application program. The program can reside in the extended virtual storage area. The parameters referenced in the call can also be in the extended virtual storage area.

Related reading: For more information about Language Environment, see *z/OS MVS Programming: Assembler Services Guide*.

COBOL compiler options for preloaded programs

If you compile your COBOL program with the COBOL for z/OS & VM compiler and preload it, you must use the COBOL compiler option RENT. Alternatively, if you compile your COBOL program with the VS COBOL II compiler and preload it, you must use the COBOL compiler options RES and RENT.

DCCTL

In a DCCTL environment, the application can only reference the address of an I/O PCB, alternate PCB, or GSAM PCB. An application program can use a PSB that contains PCBs referencing databases; however, these PCBs cannot be used during processing. Entry statements for COBOL, PL/I, C, and Pascal must refer to all PCBs included in the PSB, including PCBs which you might not be able to process, as PCBs must be included in the order in which they are listed in the PSB. This includes all PCBs prior to the last referenced PCB and can include DB PCBs. If you used a GSAM PCB, all PCBs ahead of it must be referenced.

Chapter 24. Message processing with IMS TM

IMS Transaction Manager application programs can be written in assembler language, C language, COBOL, Pascal, and PL/I to process messages.

How your program processes messages

To retrieve and send messages, an IMS TM application program issues calls to IMS TM. When your program issues a call to retrieve a message, IMS TM places the input message in the I/O area you name in the call. Before you issue a call to send a message, you must build the output message in an I/O area in your program.

Message types

An operator at a terminal can send four kinds of messages to IMS TM.

The destination of an IMS TM message identifies which kind of message is being sent:

- **Another terminal**. A logical terminal name in the first 8 bytes means that this is a message switch destined for another terminal. For a user at a logical terminal to send a message to another logical terminal, the user enters the name of the receiving logical terminal followed by the message. The IMS TM control region routes the message to the specified logical terminal. This kind of message does not result in the scheduling of any activity in a message processing program (MPP).
- **An application program**. A transaction code in the first 8 bytes means that the message is destined for an application program. IMS TM uses a transaction code to identify MPPs and transaction-oriented batch message processing programs (BMPs). To use a particular application program to process requests, the user enters the transaction code for that application program.
- **IMS TM**. A "/" (slash) in the first byte means that the message is a command destined for IMS TM.
- **Message switch service**. A system service DFSAPPC request is destined for the message switch service.

An application program can send three kinds of messages:

- Commands. A "/" in the first byte of the message text means that the message is a command for IMS TM. Programmers design applications to issue commands when they want a program to perform tasks that an operator at a terminal usually performs. This is called automated operator interface (AOI) and is described in *IMS Version 11 Communications and Connections* and *IMS Version 11 Operations and Automation*.

  Use the `CMD` call to issue commands. Do not use the `ISRT` call for issuing commands, because a message created with `ISRT` can contain a slash in the first byte without being a command.
- Messages to logical terminals by specifying a logical terminal name.
- Program-to-program switches using a transaction code.

The messages that your program receives and sends are made up of segments. Use a `GU` call to retrieve the first segment of a new message, and use `GN` calls to retrieve the remaining segments of the message. The following figure shows three

© Copyright IBM Corp. 1974, 2015

messages. Message A contains one segment, message B contains two segments, and message C contains three segments.

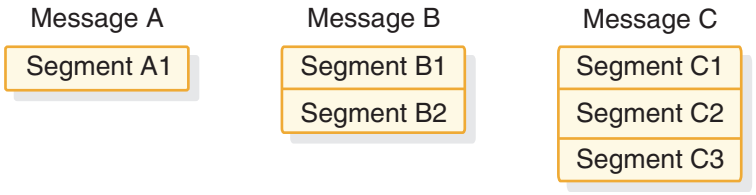


Figure 73. Message segments

To retrieve message A, you only have to issue a GU call. To retrieve messages B and C, issue one GU call to retrieve the first segment, then a GN call for each remaining segment. This assumes that you know how many segments each message contains. If you do not know this, issue GN calls until IMS TM returns a QD status code, indicating that all of the segments for that message have been retrieved.

If you inadvertently issues a GU call after retrieving the first segment of the multi-segment messages, IMS TM returns a QC status code. This status indicates that no more messages are present, without your program retrieving the additional segments associated with the message. Data would have been lost without any indication that it happened.

Input message format and contents

The input message that an application program receives from a terminal or another program always has these fields: the length field, the ZZ field, the transaction code field, and the text field.

The tables that follow show the message input layouts. The input message field names are in the first row of each table. The number below each field name is the length in bytes that has been defined for that field. The following table shows the format of an input message for the AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces. The message is slightly different for the PLITDLI interface.

Table 70. Input message format

Field Name	Field Length
LL	2
ZZ	2
TRANCODE	8
Text	Variable

Table 71. Input message format for the PLITDLI interface

Field Name	Field Length
LLLL	4
ZZ	2
TRANCODE	8
Text	Variable

The contents of the input message fields are:

LL or LLLL
The length field contains the length of the input message segment in binary, including LL (or LLLL) and ZZ. IMS TM supplies this number in the length field when you retrieve the input message.

For the AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces, define the LL field as 2 bytes long.

For the PLITDLI interface, define the LLLL field as 4 bytes long. The value in the LLLL field is the input message length minus 2 bytes. For example, if the text is 12 bytes, then the fullword LLLL contains a value of 24 bytes. This value is the total of LLLL (4 bytes) + ZZ (2 bytes) + TRANCODE (8 bytes) + text (12 bytes) - 2 bytes.

ZZ The ZZ field is a 2-byte field that is reserved for IMS TM. Your program does not modify this field.

TRANCODE
The TRANCODE is the transaction code for the incoming message.

Text
This field contains the message text sent from the terminal to the application program. The first segment of a message can also contain the transaction code associated with the program in the beginning of the text portion of the message. Input messages do not have to include the transaction code, but you can provide it for consistency.

The text field's contents in the input message and the formatting of the contents when your program receives the message depends on the editing routine your program uses.

Output message format and contents

The format of the output message that you build to send back to a terminal or to another program is similar to the format of the input message, but the fields contain different information.

Output messages contain four fields: the length field, the Z1 field, the Z2 field, and the text field. The following tables show the message output layouts. The output message field names are in the first row of each table. The number below each field name is the length in bytes that has been defined for that field. The following table shows the format of an output message for AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces. The format for PLITDLI is slightly different.

Table 72. Output message format

Field Name	Field Length
LL	2
Z1	1
Z2	1
Text	Variable

Table 73. Output message format for PLITDLI

Field Name	Field Length
LLLL	4
Z1	1

Table 73. Output message format for PLITDLI (continued)

Field Name	Field Length
Z2	1
Text	Variable

The contents of the output message fields are:

LL or LLLL
The field length contains the length of the message in binary, including the LL (or LLLL), Z1, and Z2 fields. For output message segments, supply this length when you are ready to send the message segment.

For the AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces, the LL field must be 2 bytes long. For the PLITDLI interface, the LLLL field must be 4 bytes long and contain the length of the message segment, minus 2 bytes.

Z1 The Z1 field is a 1-byte field that must contain binary zeros. It is reserved for IMS TM.

Z2 The Z2 field is a 1-byte field that can contain special device-dependent instructions (such as instructions to ring the alarm bell, instructions to disconnect a switched line, or paging instructions) or device-dependent information (such as information about structured field data or bypassing MFS).

If you do not use any of these instructions, the Z2 field must contain binary zeros. For MFS, this field contains the number of the option that is being used for this message.

Text
The text portion of the message segment contains the data that you want to send to the logical terminal or to an application program. (Text messages are typically EBCDIC characters.) The length of the text depends on the data that you want to send.

When a message is processed

A program's response to a message will depend on the type of message the program receives. A transaction code associates a request for information from a terminal with the application program that can process and respond to that request. IMS TM schedules an MPP when there are messages to be processed that contain the transaction code associated with that MPP.

Example: Suppose you have an MPP that processes the transaction code "INVINQ" for inventory inquiry. The MPP receives a request from a user at a terminal for information on the inventory of parts. When the user enters the transaction code for that application program, IMS TM schedules the application program that can process the request.

When you enter INVINQ and one or more part numbers, the MPP sends your program the quantity of each part on hand and the quantity on order.

When you enter INVINQ at the terminal, IMS TM puts the message on the message queue for the MPP that processes INVINQ. Then, after IMS TM has scheduled the MPP, the MPP issues GU and GN calls to retrieve the message. To retrieve the messages from LTERM1, the application program issues a GU for the

first segment of a message, then issues GN calls until IMS TM returns a QD status code. This means that the program has retrieved all of the segments of that message. The program then processes the request, and sends the output message to the queue for your logical terminal. (The logical terminal name is in the I/O PCB.) When the MPP sends the output message, IMS TM sends it to the queue for that logical terminal, and the message goes to the physical terminal. The following figure shows the flow of a message between the terminal and the MPP.

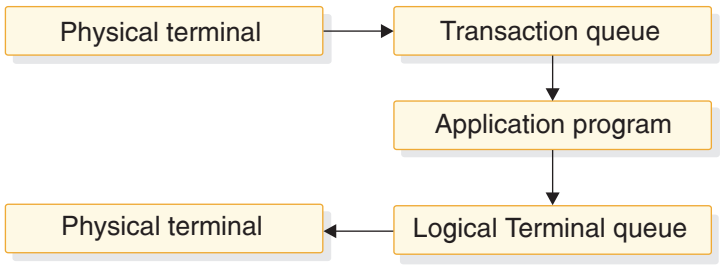


Figure 74. Transaction message flow

The following example shows the calls you use, the status codes, and what the input and output for the inventory inquiry would look like. To show you how to use GU and GN to retrieve messages, and how you insert multiple-segment messages, this example shows messages containing three segments. If input and output messages in this example were single segment messages, the program would issue only a GU to retrieve the entire message, and only one ISRT to send the message.

The message formats shown are examples; not all messages are in this format. When the program receives the input message in the I/O area, the first field of each segment contains the length of that segment. This is the LL field in the figure. For clarity, the figure shows this length in decimal; in the input message, however, it is in binary. The second field (ZZ) is reserved for IMS TM; it is 2 bytes long. The text of the message follows the reserved 2 bytes. The first message segment contains the transaction code in the 8 bytes following the ZZ field. These are the first 8 bytes of the text portion of the message.

The format of the output messages is the same. You do not need to include the name of the logical terminal, because it is in the first 8 bytes of the I/O PCB.

PART, QTY, and ON ORDER in the example are headings. These are values that you can define as constants that you want to appear on the terminal screen. To include headings in MFS output messages, define them as literals.

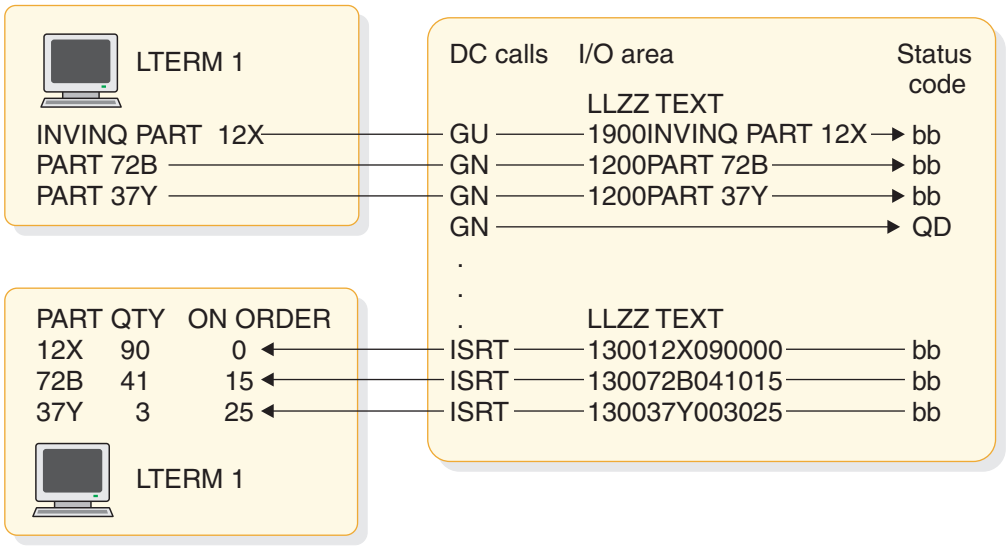


Figure 75. Inventory inquiry MPP example

Results of a message: I/O PCB

After your program issues a call, IMS TM returns information about the results of the call in the I/O PCB. To find out about the results of the call, your application program must check the information that IMS TM returns to the I/O PCB.

When your application program retrieves a message, IMS TM returns this information about the message to the I/O PCB:

- The name of the terminal that sent the message.
- A 2-character status code describing the results of the call. If the program receives a status code of QC after issuing a call to retrieve a message, no more messages are available for the program to process.
- The current date, time, and sequence number for the message.
- The user ID of the person at the terminal or the transaction code for the program that sent the message.

Because the I/O PCB resides in storage outside of your program, you define a mask of the PCB in your program based at this address to check the results of IMS TM calls. The mask contains the same fields in the same order as the I/O PCB.

Related reference:

"Specifying the I/O PCB mask" on page 379

How IMS TM edits messages

When an application program passes messages to and from a terminal, IMS TM edits the messages before the program receives the message from the terminal and before the terminal receives the message from the application program.

IMS TM gives you many choices about how you want your messages to appear both on the terminal screen and in the program's I/O area. You need to know which editing routines have been specified for your program and how they affect your programming.

The three editing routines available to non-LU 6.2 terminals in IMS TM are:

Basic Edit
Performs basic edit functions if you do not use MFS and if the message does not originate at an LU 6.1 device. You must provide control characters for some formatting functions.

Intersystem Communication (ISC) Edit
Provides the default edit for messages that originate from an LU 6.1 device. You can enter binary data in addition to text.

Message Format Service (MFS)
Formats messages through control blocks. You define the way the messages look with the control blocks.

For LU 6.2 devices, use the LU 6.2 Edit exit routine to edit input and output messages.

Related reading: For more information on LU 6.2, see *IMS Version 11 Communications and Connections*. For more information on LU 6.2 Edit exit routine, see *IMS Version 11 Exit Routines*.

Printing output messages

To print output messages, you must provide the horizontal and vertical control characters that are necessary to format your output messages.

To print your output at a printer terminal, include these control characters where necessary within the text of the message:

X'05' Skip to the tab stop, but stay on the same line.

X'15' Start a new line at the left margin.

X'25' Skip to a new line, but stay at the same place horizontally.

If you want to skip multiple lines, you can start a new line (X'15'), then skip as many lines as necessary (X'25').

Using Basic Edit

If you do not use MFS or an LU 6.1 device, IMS TM does some editing automatically. The editing IMS TM does to the first message segment is different from the editing IMS TM does for subsequent message segments.

See *IMS Version 11 Communications and Connections* for a complete description of Basic Edit.

Editing input messages

When IMS TM receives the first segment of an input message for your application program, IMS TM:

- Removes leading and trailing control characters.
- Removes leading blanks.
- Removes backspaces (from a printer terminal).
- Translates to uppercase, if this is specified with the EDIT=UC specification on the system definition TRANSACT macro.

If the message segment contains a password, IMS TM edits the segment by:

- Removing the password and inserting a blank in place of the password.
- Removing the password if the first character of the text is a blank. IMS TM does not insert the blank.
- Left-justifying the text of the segment.

For subsequent input message segments, IMS TM does not remove leading blanks from the text of the message. The other formatting features are the same.

Editing output messages

For output messages, Basic Edit:

- Changes nongraphic characters in the output message before the data goes to the output device.
- Inserts any necessary idle characters after new line, line feed, and tab characters.
- Adds line control characters for the operation of the communication line.

Using Intersystem Communication Edit

Intersystem Communication (ISC) Edit is the default edit for messages from LU 6.1 devices. It is not valid for any other device types. One advantage of using ISC edit is that IMS TM does not edit the text of a message, allowing you to enter binary data.

Editing input messages

The editing IMS TM does to input messages depends on whether the Function Management (FM) header contains the SNA-defined primary resource name (PRN) parameter. In either case, IMS TM removes the FM header before the input message is received by the application program.

If the FM header does not contain the PRN parameter:

- IMS TM removes leading control characters and blanks when it receives the first segment of an input message for your application program.
- If the message segment contains a password, IMS TM removes the password and inserts a blank where the password was.
- IMS TM does not edit the text of the message (the data following the password).

If the FM header contains the PRN parameter:

- The PRN is treated as the transaction code and is received by your application program as the first field in the message segment.
- The message segment is not edited by IMS TM.

Editing output messages

ISC edit does not edit output messages.

Using Message Format Service

Format the messages that you send to MPP using the Message Format Service (MFS). You define the format in control blocks.

The MFS control blocks indicate to IMS TM how you want your input and output messages arranged:

- For input messages, MFS control blocks define how the message that the terminal sends to your MPP is arranged in the I/O area.
- For output messages, MFS control blocks define how the message that your MPP sends to the terminal is arranged on the screen or at the printer. You can also define words or other data that appear on the screen (headings, for example) but do not appear in the program's I/O area. This data, called a literal, can be a field in the output message from the application program or a field in the input message from the terminal.

Terminals and MFS

Whether your program uses MFS depends on the types of terminals and secondary logical units (SLUs) your network uses. You can bypass MFS formatting of an output message for a 3270 device or for SLU Type 2 devices. When MFS is bypassed, you construct the entire 3270 data stream from within your program.

Restriction: MFS cannot be used with LU 6.2 devices (APPC).

Related reading: For more information on LU 6.2 and APPC, see *IMS Version 11 Communications and Connections*.

Using MFS involves high-level design decisions that are separate from the tasks of application design and application programming; many installations that use MFS have a specialist who designs MFS screens and message formats for all applications that use MFS.

MFS makes it possible for an MPP to communicate with different types of terminals without having to change the way it reads and builds messages. When the MPP receives a message from a terminal, the message's format in the MPP I/O area depends on the MFS options specified and not on what kind of terminal sent it. MFS shields the MPP from the physical device that is sending the message in the same way that a DB PCB shields the program from what the data in the database actually looks like and how it is stored.

MFS input message formats

You define a message to MFS in fields just as you would define fields within a database segment.

When you define the fields that make up a message segment, you give MFS information such as:

- The field length
- The fill character used when the length of the input data is less than the length defined for the field
- Whether the data in the field is left-justified or right-justified
- If the field is truncated, whether it is truncated on the left or right

The order and length of these fields within the message segment depends on the MFS option that your program is using. You specify the MFS option in the MID. The decision of which option to use for an application program is based on:

- How complex the input data is
- How much the input data varies
- The language the application program is written in
- The complexity of the application program
- Performance factors

The Z2 field in MFS messages contains the MFS formatting option being used to format the messages to and from your program. If something is wrong in the way that IMS TM returns the messages to your I/O area, and you suspect that the problem might be with the MFS option used, you can check this field to see if IMS TM is using the correct option. A X'00' in this field means that MFS did not format the message at all.

One way to understand how each of the MFS options formats your input and output messages is to look at examples of each option.

Example: Suppose that you have defined the four message segments shown in the following table. Each of the segments contains a 2-byte length field and a 2-byte ZZ field. The first segment contains the transaction code that the person at the terminal entered to invoke the application program. The number of bytes defined for each field appears below the name of the field in the figure.

When you use the PLITDLI interface, you must define the length field as a binary fullword, LLLL. When you use the AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, or PASTDLI interfaces, you must define the length field as a halfword, LL. The value provided by the PL/I application program must represent the actual segment length minus 2 bytes. For example, if the output text is 10 bytes, then the value of the fullword LLLL is 14 and is the sum of the length of LLLL (4 bytes - 2 bytes) + Z1 (1 byte) + Z2 (1 byte) + TEXT (10 bytes).

Table 74. Four-segment message.

Segment Number	Field Name	Field Length	Field Value
1	LL	2	0027
	ZZ	2	XXXX
	TRANCODE	8	YYYY
	Text	5	PATIENT#
	Text	10	NAME
2	LL	2	0054
	ZZ	2	XXXX
	Text	50	ADDRESAF
3	LL	2	0016
	ZZ	2	XXXX
	Text	6	CHARGES
	Text	6	PAYMENTS
4	LL	2	0024
	ZZ	2	XXXX
	Text	10	TREATMENT
	Text	10	DOCTOR

For these examples, assume that:

- The transaction code is defined in the MID as a literal.
- All of the fields are left-justified.
- The fill character is defined as a blank. When the length of the data in a field is less than the length that has been defined for that field, MFS pads the field with fill characters. Fill characters can be:

– Blanks
– An EBCDIC character
– An EBCDIC graphic character
– A null, specified as X'3F'

When you specify that the fill character is to be a null, MFS compresses the field to the length of the data if that length is less than the field length.

The fields for segment 4 of the message in the previous table are arranged on the terminal screen in the format shown in the following figure.

Example: Assume the person enters the name of a patient, and the charges and payments associated with that patient.

```
PATIENT#:          NAME:  MC ROSS

ADDRESAF:

CHARGES: 106.50    PAYMENTS:  90.00

TREATMENT:

DOCTOR:
```

Figure 76. Terminal screen for MFS example

MFS provides three options for message formatting.

MFS option 1

Use this option when the program receives and transmits most of the fields in the message segments. The way that option 1 formats messages depends on whether you have defined a null as the fill character for any of the fields in the segment.

If none of the fields in the message were defined as having a fill character of null:

- The program receives all the segments in the message.
- Each segment is the length that was specified for it in the MID.
- Each segment contains all its fields.
- Each field contains data, data and fill characters, or all fill characters.

The following table shows the Option 1 Format of segments received by the application program.

Table 75. MFS option 1 message format

Segment Number	Field Name	Field Length	Field Value
1	LL	2	0027
	Z1	1	XX
	Z2	1	01
	TRANCODE	8	YYYY
	Text	5	blanks
	Text	10	MCROSSbbbb

Table 75. MFS option 1 message format (continued)

Segment Number	Field Name	Field Length	Field Value
2	LL	2	0054
	Z1	1	XX
	Z2	1	01
	Text	50	blanks
3	LL	2	0016
	Z1	1	XX
	Z2	1	01
	Text	6	010650
	Text	6	009000
4	LL	2	0024
	Z1	1	XX
	Z2	1	01
	Text	10	blanks
	Text	10	blanks

The message format for option 1 output messages is the same as the input message format. The program builds output messages in an I/O area in the format shown for segment 4 in the previous figure. The program can truncate or omit fields in one of two ways:

- Inserting a short segment
- Placing a null character in the field

If one or more of the fields are defined as having a null fill character, the message is different. In this case, the message has these characteristics:

- If a field has been defined as having a fill character of null and the terminal offers not data, the field is eliminated from the message segment.
- If all of the fields in a segment have a null fill character and none of the fields contains any literals, the segment is eliminated from the message.
- If only some of the fields in a segment have a null fill character, any field containing nulls is eliminated from the segment. The relative positions of the fields remaining within the segments are changed.
- When the length of the data that is received from the originating terminal is less than the length that is been defined for the field, the field is truncated to the length of the data.

MFS option 2

Use this option when the program processes multisegment messages where most of the fields are transmitted but some of the segments are omitted. Option 2 formats messages in the same way that option 1 does, unless the segment contains no input data from the terminal after IMS TM has removed the literals. If this is true, and if no additional segments in the message contain input data from the terminal, IMS TM ends the message. The last segment that the program receives is the last segment that contains input data from the terminal.

Sometimes a segment that does not have any input data from the terminal is followed by segments that do contain input data from the terminal. When this happens, MFS gives the program the length field and the Z fields for the segment, followed by a 1-byte field containing X'3F'. This indicates to the program that this is a null segment.

If the message segments shown in Table 74 on page 402 are formatted by option 2, they appear in the format shown in the table below.

Table 76. MFS option 2 message format

Segment Number	Field Name	Field Length	Field Value
1	LL	2	0027
	Z1	1	XX
	Z2	1	02
	TRANCODE	8	YYYY
	Text	5	blanks
	Text	10	MCROSSbbbb
2	LL	2	0005
	Z1	1	XX
	Z2	1	02
	Text	1	X'3F'
3	LL	2	0016
	Z1	1	XX
	Z2	1	02
	Text	6	010650
	Text	6	009000

Segment 2 in the previous table contains only a X'3F' because that segment is null, but Segment 3 contains data. This message does not contain a segment 4 because it is null.

MFS option 3

Use this option when the program receives and transmits only a few of the fields within a segment. When you use option 3, the program receives only those fields that have been received from the terminal. The program receives only segments that contain fields received from the originating terminal. Segments and fields can be of variable length if you have defined option 3 as having a null fill character.

A segment in an option 3 message is identified by its relative segment number—in other words, what position in the message it occupies. The fields within a segment are identified by their offset count within the segment.

Example: The NAME field in segment 1 is (MCROSSbbbb. The value 17 is the sum of the lengths of the fields preceding the NAME field and includes an 8-byte transaction code and a 5-byte field of blanks. It does not include the 2-byte relative segment number field (field A in the following table), the 2-byte length field (field B), or the 2-byte relative offset field (field C).

Option 3 messages do not contain literals defined in the MID. This means that the transaction code is removed from the message, except during a conversation. If the transaction that the program is processing is a conversational transaction, the transaction code is not removed from the message. The transaction code still appears in the scratchpad area (SPA).

Each segment the program receives contains the relative number of this segment in the message (field A in the following table). In addition, each data field within the segment is preceded by two fields:

- A 2-byte length field (B). Including the length field itself, the 2-byte relative field offset, and the data in the field.
- A 2-byte relative field offset (C), giving the field's position in the segment as defined in the MID.

These two fields are followed by the data field. MFS includes these fields for each field that is returned to the application program.

If the message segments shown in Table 74 on page 402 are formatted by option 3, they appear in the format shown in the following table. The notes for the tables explain the letters A, B, C, and D, which are in the first row of segment 1 and segment 3.

Table 77. MFS option 3 message format

Segment Number	Field Name	Field Length	Field Value
1	LL	2	0020
	Z1	1	XX
	Z2	1	03
	A	2	0001
	B	2	0014
	C	2	0017
	D	10	MCROSSbbbb
2	LL	2	0000
	Z1	1	XX
	Z2	1	03
	A	2	0003
	B	2	0010
	C	2	0004
	D	6	010650
	B	2	0010
	C	2	0010
	D	6	009000

Notes to the previous table:

- The fields marked A contain the relative segment number. This number gives the segment's position within the message.
- The fields marked B contain the field length. This length is the sum of the lengths of B field (2 bytes) + C field (2 bytes) + D field (the length of the data).

- The fields marked C contain the relative field offset. This gives each field's position within the segment.
- The fields marked D contain the data from the terminal. In this example, the fill character was defined as blank, so the data field is always its defined length. IMS TM does not truncate it. If you define the fill character as null, the lengths of the data fields can differ from the lengths defined for them in the segment. With a null fill character, if the length of the data from the terminal is less than the length defined for the field, IMS TM truncates the field to the length of the data. Using a null fill with option 3 reduces the space required for the message even further.

MFS output message formats

The output message format is used to define what segments and fields MFS will receive from the application program.

If using option 1 or option 2, the output message format is the same as it is for input messages. Present all fields and segments to MFS. You can present null segments. All fields in output messages are fixed length and fixed position. Output messages do not contain option numbers.

Option 3 output messages are similar to input messages, except that they do not contain option numbers. The program submits the fields as required in their segments with the position information.

Using LU 6.2 User Edit exit routine (optional)

This exit routine edits input and output messages from LU 6.2 devices when the implicit application program interface support is used.

If it is not provided, then messages are presented without modification. IMS does not invoke the exit for CPI-C driven transactions because IMS does not participate in the data flows when the application program uses the CPI directly.

The LU 6.2 User Edit exit routine is called once for each message segment or inbound control flow. You can call the exit routine for data messages and use it to:

- Examine the contents of a message segment.
- Change the contents of a message segment.
- Expand or compact the contents of a message segment.
- Discard a message segment and process subsequent segments, if any.
- Use the Deallocate_Abend command to end the conversation.

For more information on LU 6.2 User Edit exit routine, see*IMS Version 11 Communications and Connections* and *IMS Version 11 Operations and Automation*.

Message processing considerations for DB2

For the most part, the message processing function of a dependent region that accesses DB2 databases is similar to that of a dependent region that accesses only DL/I databases.

The method each program uses to retrieve and send messages and back out database changes is the same. The differences are:

- DL/I statements are coded differently from SQL (structured query language) statements.

- When an IMS TM application program receives control from IMS TM, IMS has already acquired the resources the program is able to access. IMS TM schedules the program, although some of the databases are not available. DB2 does not allocate resources for the program until the program issues its first SQL statement. If DB2 cannot allocate the resources your program needs, your program can optionally receive an initialization error when it issues its first SQL call.
- When an application issues a successful checkpoint call or a successful message GU call, DB2 closes any cursors that the program is using. This means that your program should issue its OPEN CURSOR statement after a checkpoint call or a message GU.

IMS TM and DB2 work together to keep data integrity in these ways:

- When your program reaches a commit point, IMS TM makes any changes that the program has made to DL/I databases permanent, releases output messages for their destinations, and notifies DB2 that the program has reached a commit point. DB2 then makes permanent any changes that the program has made to DB2 databases.
- When your program terminates abnormally or issues one of the IMS TM rollback calls (ROLB, ROLS without a token, or ROLL), IMS TM cancels any output messages your program has produced, backs out changes your program has made to DL/I databases since the last commit point, and notifies DB2. DB2 backs out the changes that the program has made to DB2 databases since the last commit point.

Through the Automated Operator Interface (AOI), IMS TM application programs can issue DB2 commands and IMS TM commands. To issue DB2 commands, the program issues the IMS TM /SSR command followed by the DB2 command. The output of the /SSR command is routed to the master terminal operator (MTO).

Sending messages to other terminals and programs

When an application program processes a message from a terminal, it usually sends the response to the terminal that sent the input message. But sometimes you might want to send output messages to a terminal other than the originating terminal, or to other terminals in addition to the originating terminal. You might also want to send messages to other application programs.

When you use an alternate PCB:

- If you want to send output messages to one alternate destination, define the alternate PCB for that destination.
- If you want to send output messages to more than one alternate destination, and you want to be able to change the destination of the alternate PCB, define the alternate PCB as modifiable during program specification block (PSB) generation. Then, before you issue the ISRT call, you issue a CHNG call to set the destination of the alternate modifiable PCB for the destination program or terminal.

  The *express alternate PCB* is a special kind of alternate PCB that is defined during PSB generation, by specifying EXPRESS=YES.

  When you use an express alternate PCB, messages you send using that PCB are sent to their final destinations immediately. Messages sent with other PCBs are sent to temporary destinations until the program reaches a commit point. Messages sent with express PCBs are sent if the program subsequently terminates abnormally, or issues one of the rollback calls: ROLL, ROLB, or ROLS.

Using an express alternate PCB in this kind of situation is a way to ensure that the program can notify the person at the terminal, even if abnormal termination occurs. For all PCBs, when a program abnormally terminates or issues a ROLL, ROLB, or ROLS call, messages inserted but not made available for transmission are cancelled, while messages made available for transmission are never cancelled.

For a nonexpress PCB, the message is not made available for transmission to its destination until the program reaches a commit point. The commit point occurs when the program terminates, issues a CHKP call, or requests the next input message and the transaction has been defined with MODE=SNGL.

For an express PCB, when IMS TM knows that it has the complete message, it makes the message available for transmission to the destination. In addition to occurring at a commit point, this also occurs when the application program issues a PURG call using that PCB or requests the next input message.

A PSBGEN can also specify an alternate PCB as an alternate response PCB defined during PSB generation.

- If you want to send a message to an LU 6.2 device, you can specify the LU 6.2 descriptor name that is associated with that device. IMS internally performs the uppercase translation of the destination name (CNT or SMB).

Related reference:

"Specifying the alternate PCB mask" on page 383

Sending messages to other terminals

To reply to a different terminal, also use the ISRT call, but use an alternate program communication block (PCB) instead of the TP PCB.

Just as the TP PCB represents the terminal that sent the message, an alternate PCB represents the terminal to which you want to send the message.

Single alternate terminal

If you are going to send messages to only one alternate terminal, you can define the alternate PCB for that terminal during PSB generation. When you define an alternate PCB for a particular destination, you cannot change that destination during program execution. Each time you issue an ISRT call that references that PCB, the message goes to the logical terminal whose name was specified for the alternate PCB. To send a message to that terminal, place one message segment at a time in the I/O area, and issue an ISRT call referring to the alternate PCB, instead of the TP PCB.

Several alternate terminals

To send messages to several terminals, you can define the alternate PCB as modifiable during PSB generation. Therefore, the alternate PCB represents more than one alternate terminal. You can change the destination while your program is running.

Before you can set or change the destination of an alternate PCB, you must indicate to IMS TM that the message you have been building so far with that PCB is finished. To do this, issue a PURG call.

PURG allows you to send multiple output messages while processing one input message. When you do not use PURG, IMS TM groups message segments into a message and sends them when the program issues a GU for a new message, terminates, or reaches a commit point. A PURG call tells IMS TM that the message

built against this TP PCB or alternate PCB (by issuing one ISRT call per message segment) is complete. IMS TM collects the message segments that you have inserted into one PCB as one message and sends it to the destination represented by the alternate PCB you have referenced.

A PURG call that does not contain the address of an I/O area indicates to IMS TM that this message is complete. If you include an I/O area in the call, PURG acts as an ISRT call as well. IMS TM treats the data in the I/O area as the first segment of a new message. When you include an I/O area on a PURG call, you can also include a MOD name to change the format of the screen for this message. Although specifying the MOD name is optional, when you use it, you can specify it only once per message or in only the first ISRT or PURG that begins the message.

To set the destination of a modifiable alternate PCB during program execution, you use a CHNG call. When you issue the CHNG call you supply the name of the logical terminal to which you want to send the message. The alternate PCB you use then remains set with that destination until you do one of the following:

- Issue another CHNG call to reset the destination.
- Issue another GU to the message queue to start processing a new message. In this case, the name still appears in the alternate PCB, even though it is no longer valid.
- Terminate your program. When you do this, IMS TM resets the destination to blanks.

The first 8 bytes of the alternate PCB contain the name of the logical terminal to which you want to send the message.

When you issue a CHNG call, give IMS TM the address of the alternate PCB you are using and the destination name you want set for that alternate PCB.

When you use the PURG call, you give IMS TM only the address of the alternate PCB. IMS TM sends the message you have built using that PCB.

To indicate an error situation, you can send a message by issuing an ISRT call followed by a PURG call against an express PCB. These calls send the message to its final destination immediately.

Example: The program could go through these steps:

1. The program issues a GU call (and GN calls, if necessary) to retrieve an input message.
2. While processing the message, the program encounters an abnormal situation.
3. The program issues a PURG call to indicate to IMS TM the start of a new message.
4. The program issues a CHNG call to set the destination of an express PCB to the name of the originating logical terminal. The program can get this name from the first 8 bytes of the I/O PCB.
5. The program issues ISRT calls as necessary to send message segments. The ISRT calls reference the express PCB.
6. The program issues a PURG call referencing the express PCB. IMS TM then sends the message to its final destination.
7. The program can then terminate abnormally, or it can issue a ROLL, ROLB, or ROLS call to back out its database updates and cancel the output messages it has created since the last commit point.

If your output messages contained three segments, and you used the PURG call to indicate the end of a message (and not to send the next message segment), you could use this call sequence:

```
CHNG ALTPCB1, LTERMA
ISRT ALTPCB1, SEG1
ISRT ALTPCB1, SEG2
ISRT ALTPCB1, SEG3
PURG ALTPCB1
CHNG ALTPCB1, LTERMB
ISRT ALTPCB1, SEG4
ISRT ALTPCB1, SEG5
ISRT ALTPCB1, SEG6
```

Sending messages to other IMS application programs

A program-to-program switch occurs when an IMS application running in an IMS dependent region sends a message to another IMS application running in an IMS dependent region.

You can issue a program-to-program switch to send and receive messages with any of the following types of IMS applications:

- message processing program (MPP)
- batch message processing (BMP) program
- Java message processing (JMP) program
- Java batch processing (JBP) program

To send a message to another online program, use an alternate program communication block (PCB) in a similar way as when sending messages to alternate terminals. If you send messages to only one application program, then you can define the alternate PCB with the transaction code for that application program during PSB generation. If you send messages to more than one application program, you can define the alternate PCB as modifiable.

If you use an alternate modifiable PCB, IMS TM makes a security check when you issue the CHNG call to set the destination of the alternate modifiable PCB. The terminal that enters the transaction code that causes the message switch must be authorized to enter the transaction code that the CHNG call places in the alternate modifiable PCB. IMS TM does not check for security when you issue the ISRT call.

When an IMS TM application program issues a CHNG call, the Resource Access Control Facility (RACF) is invoked and a check is made to determine whether the originating terminal is authorized for the transaction code that was issued. If, instead of using the CHNG call, the program issues an ISRT call against a preset alternate PCB, no security check is made, regardless of the environment.

When you issue a program-to-program message switch, you have the same considerations as when you communicate with a logical terminal. Keep in mind the following points:

- Create an I/O area large enough to hold the largest segment that you are sending.
- Use an alternate PCB, not the TP PCB, to send the message.
- Issue a CHNG call before the ISRT call to place the transaction code of the program in the first field of the alternate PCB. If the alternate PCB was set to this transaction code in the PSBGEN, issue the ISRT call.
- IMS TM must know the transaction code. Define it at system definition.

- A nonconversational program can do a program-to-program message switch to another nonconversational program, but not to a conversational program.
- A conversational program can do a program-to-program message switch to either another conversational program or a nonconversational program.

Open Transaction Manager Access (OTMA) program-to-program switching has the following restrictions:

- In a shared queues environment that has both synchronous APPC/OTMA support (AOS=Y on the DFSDCxxx PROCLIB member) and RRS support (RRS=Y on the startup procedure) enabled, an application program running on a back-end IMS system that initiates an outbound APPC protected conversation with another IMS system is restricted to a single program-to-program switch.
- If an application program performs multiple program-to-program switches after allocating an APPC outbound protected conversation on another IMS system, the results are unpredictable and can include a WAIT-RRS/PC condition in the message processing region (MPR).

A message switch to another conversational program transfers the scratchpad area (SPA) and the responsibility to respond to the originating terminal to the new application program. A message switch to a nonconversational program does not change the responsibilities of the conversational program. The conversational program must still return the SPA to IMS TM (if the SPA has been modified) and must respond to the originating terminal. The following tables show the format for an output message to an application program.

Table 78. Message Format for program-to-program message switch for AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces

Field Name	Field Length
LL	2
Z1	1
Z2	1
Text	Variable

Table 79. Message format for program-to-program message switch for the PLITDLI interface

Field Name	Field Length
LLLL	4
Z1	1
Z2	1
Text	Variable

The format is the same as for output messages to terminals. Z1 and Z2 are fields that must contain binary zeros. These fields are reserved for IMS. The text field contains the message segment that you want to send to the application program.

If the program that is processing the message expects the transaction code, include the transaction code of the recipient program as part of the message text of the first segment of the message, because IMS TM does not automatically include the transaction code in the first segment of a switched message. Including the transaction code in the message text of the first segment keeps the first segments of all messages in the same format, regardless of whether they are sent from terminals or other programs.

Related concepts:

"Passing the conversation to another conversational program" on page 423

Related tasks:

"Program switching in JMP and JBP applications" on page 662

How the VTAM I/O facility affects your VTAM terminal

VTAM terminals can fail to respond to requests sent by IMS. The master terminal operator or an automated operator interface application program can optionally activate a "timeout" facility. This allows a message stating a specific amount of time has passed to be sent to the master terminal operator.

IMS TM can be set up to do one of the following:

- Do nothing, which means that your terminal remains inactive. This is the default.
- Send a message to the master terminal operator stating that the specified period of time has passed. The operator can then determine what action, if any, should be taken.
- Send a message to the master terminal operator stating that the specified period of time has passed. IMS TM then issues the VTAM `VARY NET, INACT` command followed by a VTAM `VARY NET, ACT` command. If the terminal is defined to IMS TM as non-shared and operable, and if IMS TM is not shutting down, IMS TM issues an OPNDST for the terminal.

  Restriction: This option does not apply to ISC terminals. If your installation chooses this option and an ISC terminal times out, a message is sent to the master terminal stating that the specified period of time has passed. The operator can determine what action, if any, should be taken.

Communicating with other IMS TM systems using Multiple Systems Coupling

In addition to communicating with programs and terminals in your IMS TM system, your program can communicate with terminals and programs in other IMS TM systems through Multiple Systems Coupling (MSC).

MSC makes this possible by establishing links between two or more separate IMS TM systems. The terminals and transaction codes within each IMS TM system are defined as belonging to that system. Terminals and transaction codes within your system are called "local," and terminals and transaction codes defined in other IMS TM systems connected by MSC links are called "remote."

Related reading: For an overview of MSC, see *IMS Version 11 Communications and Connections*.

Implications of MSC for program coding

For the most part, communicating with a remote terminal or program does not affect how you code your program. MSC handles the message routing between systems.

For example, if you receive an input message from a remote terminal, and you want to reply to that terminal, you issue an `ISRT` call against the I/O PCB—just as you would reply to a terminal in your system.

In the following two situations, MSC might affect your programming:

- When your program needs to know whether an input message is from a remote terminal or a local terminal. For example, if two terminals in separate IMS TM systems had the same logical terminal name, your program's processing might be affected by knowing which system sent the message.
- When you want to send a message to an alternate destination in another IMS TM system.

Restriction: If a transaction allocated by an LU 6.2 device is destined to a remote system through MSC links, IMS rejects the transaction with the message `TP_NOT_Avail_No_Retry`.

Directed routing makes it possible for your program to find out whether an input message is from your system or from a remote system, and to set the destination of an output message for an alternate destination in another IMS TM system. With directed routing, you can send a message to an alternate destination in another IMS TM system, even if that destination is not defined in your system as remote.

Restriction: MSC directed routing does not support a program-to-program switch between conversational transactions.

Related Reading: For more information about LU 6.2 and about MSC directed routing, see *IMS Version 11 Communications and Connections*.

Receiving messages from other IMS TM systems

When an application program retrieves an input message, the program can determine whether the input message is from a terminal or program in its IMS TM system, or from a terminal or program in another IMS TM system. There might be situations in which the application program's processing is changed if the input message is from a remote terminal, rather than from a local terminal.

For example, suppose that your IMS TM system is system A, and that it is linked to another IMS TM system called system B. MSC links are one-way links. The link from system A to system B is called LINK1, and the link from system B to system A is called LINK2. The application program named MPP1 runs in system A. The logical terminal name of the master terminals in both systems is MASTER. The following figure shows systems A and B.

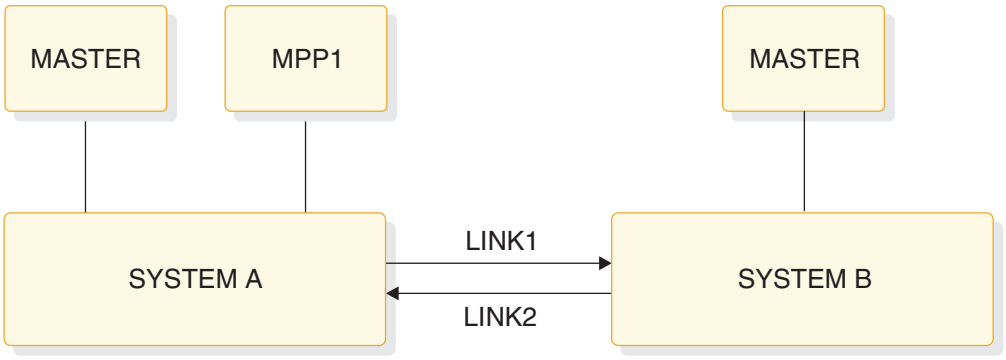


Figure 77. MSC example

If the MASTER terminal in system B sends a message indicating that the system is shutting down to MPP1 in system A, MPP1 needs to know that the message is from MASTER in system B and not MASTER in system A.

If you have specified ROUTING=YES on the TRANSACT macro during IMS TM system definition, IMS TM does two things to indicate to the program that the message is from a terminal in another IMS TM system.

First, instead of placing the logical terminal name in the first field of the I/O PCB, IMS TM places the name of the MSC logical link in this field. In the example, this is LINK1. This is the logical link name that was specified on the MSNAME macro at system definition. However, if the message is subsequently sent back to the originating system, the originating LTERM name is reinstated in the first field of the I/O PCB.

Second, IMS TM turns on a bit in the field of the I/O PCB that is reserved for IMS. This is the second bit in the first byte of the 2-byte field. The following figure shows the location of this bit within the reserved field.

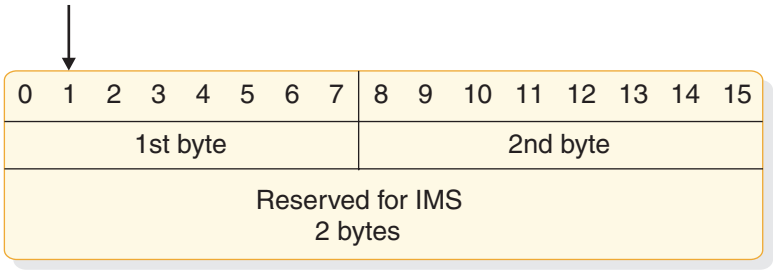


Figure 78. Directed routing bit in I/O PCB

MPP1 tests this bit to determine if the message is from MASTER in system A. If it is, MPP1 should terminate immediately. However, if the message is from MASTER in system B, MPP1 could perform some local processing and send transactions for system B to a message queue so that those transactions could be processed later on, when system B is up.

Sending messages to alternate destinations in other IMS TM systems

To send an output message to an alternate terminal in another IMS TM system, your system must have an MSC link with the system to which you want to send the message.

To do this, issue a `CHNG` call against an alternate PCB and supply the name of the MSC link (in the example this is LINK1) that connects the two IMS TM systems.

For example, if you were sending a message to TERMINAL 1 in system B after you received a message from some other terminal, you would first issue this `CHNG` call:

```
CHNG altpcb, LINK1
```

Then issue an `ISRT` call (or calls) to send the message just as you would send a message to a local terminal. The following tables show the format of the Direct Routing Output Message.

Table 80. Directed routing output message format for AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces

Field Name	Field Length
LL	2

Table 80. Directed routing output message format for AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces (continued)

Field Name	Field Length
ZZ	2
DESTNAME	1 - 8
b	1
Text	Variable

Table 81. Directed routing output message format for the PLITDLI interface

Field Name	Field Length
LLLL	4
ZZ	2
DESTNAME	1 - 8
b	1
Text	Variable

The field formats in a directed routing output message are listed below:

- The LL and ZZ fields are 2 bytes each (For the PLITDLI interface, use the 4-byte field LLLL). LL (or LLLL) contains the total length of the message. This is the sum of all of the fields in the message, including the LL field (and in PL/I, LLLL contains the total length minus 2). ZZ is reserved for IMS.
- The destination name, DESTNAME, is the name of the logical terminal to which you are sending the message. This field is from 1 to 8 bytes long and it must be followed by a blank.

  If the destination in the other system is a terminal, IMS TM removes the DESTNAME from the message. If the destination in the other system is a program, IMS TM does not remove the DESTNAME.
- The TEXT field contains the text of the message. Its length depends on the message you are sending.

If your message contains a security violation, MSC detects it in the receiving system (in this case, system B), and reports it to the person at the originating terminal (system A).

IMS conversational processing

You can write conversational programs to perform conversational processing with IMS Transaction Manager.

The difference between a conversational and nonconversational program is:

Conversational program
: A message processing program (MPP) that processes transactions made up of several steps. It does not process the entire transaction at the same time. A conversational program divides processing into a connected series of terminal-to-program-to-terminal interactions. You use conversational processing when one transaction contains several parts.

Nonconversational program
: A message processing program that receives a message from a terminal, processes the request, and sends a message back to the terminal. A

conversational program receives a message from a terminal, and replies to the terminal, but saves the data from the transaction in a scratchpad area (SPA). Then, when the person at the terminal enters more data, the program has the data it saved from the last message in the SPA, so it can continue processing the request without the person at the terminal having to enter the data again.

A conversational example

The following example shows how to use conversational processing to find out if a customer can qualify for a car loan.

This inquiry contains two parts. First, you give the name and address of the person requesting the loan and the number of years for which the person wants the loan. After you provide this information, IMS TM asks you for the information on the car: model, year, and cost. You enter this information, IMS TM invokes the program that processes this information, and the program tells you whether the loan can be granted.

If you use MFS, the process involves these steps:

1. Enter the format command (/FORMAT) and the MOD name. This tells IMS to format the screen in the way defined by this message output descriptor (MOD).

   If the MOD name is CL, the command is:

   ```
   /FORMAT CL
   ```

   IMS TM then takes that MOD from the MFS library and formats your screen in the way defined by the MOD. When the MOD for the car loan application formats your screen, it looks like this:

   ```
   CARLOAN
   NAME:
   ADDRESS:
   YEARS:
   ```

   The word “CARLOAN” is the transaction code for this application. Each transaction code is associated with an application program, so when IMS TM receives the transaction code “CARLOAN”, IMS TM knows what application program to schedule for this request.
2. Enter the customer's name and address, and the length of the loan. When you enter this information, your screen looks like this:

   ```
   CARLOAN
   NAME:    JOHN EDWARDS
   ADDRESS: 463 PINEWOOD
   YEARS:   5
   ```
3. IMS TM reads the transaction code, CARLOAN, and invokes the program that handles that transaction code. MFS formats the information from the screen for the MPP's I/O area by using the DIF and the MID.

   When the MPP issues its first call, which is usually a GU for the SPA, IMS TM clears the SPA to binary zeros and passes it to the application program.
4. Next, the MPP processes the input data from the terminal and does two things. It moves the data that it will need to save to the SPA, and it builds the output message for the terminal in the I/O area. The information that the MPP saves in the SPA is the information the MPP will need when the second part of the request comes in from the terminal. You do not save information in the SPA that you can get from the database. In this example, you save the name of the customer applying for the loan, because if the customer is granted the loan, the program uses the customer name to locate the information to be updated in the database.

The program then issues an ISRT call to return the SPA to IMS, and another ISRT call to send the output message to the terminal.

The response that the MPP sends to the terminal gives IMS TM the name of the MOD to format the screen for the next cycle of the conversation. In that cycle, you need to supply the model, year, and cost of the car that John Edwards wants to buy. Your screen looks like this:

```
MODEL:
YEAR:
COST:
```

5. IMS TM again uses the device input format (DIF) and message input descriptor (MID) associated with the transaction code, and sends the information back to the MPP. The MPP has not been running all this time. when IMS TM receives the terminal input with the transaction code CARLOAN, IMS TM invokes the MPP that processes that transaction again for this cycle of the conversation.
6. IMS TM returns the updated SPA to the MPP when the MPP issues a GU, then returns the message to the MPP when the MPP issues a GN. The MPP does the required processing (in this case, determining whether the loan can be granted and updating the database if necessary), and is then ready to end the conversation. To do this, the MPP blanks out the transaction code in the SPA, inserts it back to IMS, then sends a message to the terminal saying whether the loan can be granted.

Conversational structure

Structuring your conversational program depends on the interactions between your program and the person at the terminal.

Before structuring your program, you need to know:

- What should the program do in an error situation?

  When a program in a conversation terminates abnormally, IMS TM backs out only the last cycle of the conversation. A cycle in a conversation is one terminal/program interaction. Because the conversation can terminate abnormally during any cycle, you should be aware of some things you can do to simplify recovery of the conversation:

  – The ROLB or ROLS call can be used in conversational programs to back out database updates that the program has made since the last commit point. ROLL can also be used in conversational programs, but terminates the conversation.

  – If possible, updating the database should be part of the last cycle of the conversation so that you do not have different levels of database updates resulting from the conversation.

  – If your program encounters an error situation and it has to terminate, it can use an express alternate (program communication block) PCB to send a message to the originating terminal, and, if desired, to the master terminal operator.

    To do this, the program issues a CHNG call against the express alternate PCB and supplies the name of the logical terminal from the TP PCB, then an ISRT call that references that PCB and the I/O area that contains the message. The program can then issue another CHNG call to set the destination of the express alternate PCB for the master terminal, and another ISRT call that references that PCB, and the I/O area that contains the output message.

- Does your application program process each cycle of the conversation?

  A conversation can be processed by one or several application programs. If your program processes each stage of the conversation (in other words, your program

processes each input message from the terminal), the program has to know what stage of the conversation it is processing when it receives each input message.

When the person at the terminal enters the transaction code that starts the conversation, IMS TM clears the SPA to binary zeros and passes the SPA to the program when the program issues a GU call. On subsequent passes, however, the program has to be able to tell which stage of the conversation it is on so that it can branch to the section of the program that handles that processing.

One technique that the program can use to determine which cycle of the conversation it is processing is to keep a counter in the SPA. The program increments this counter at each stage of the conversation. Then, each time the program begins a new cycle of the conversation (by issuing a GU call to retrieve the SPA), the program can check the counter in the SPA to determine which cycle it is processing, then branch to the appropriate section.

- How can your program pass control of the conversation to another conversation program?

  Sometimes it is more efficient to use several application programs to process a conversation. This does not affect the person at the terminal. It depends on the processing that is required.

  In the car loan example, one MPP could process the first part of the conversation (processing the name, address, and number of years), and another MPP could process the second part of the conversation (processing the data about the car and responding with the status of the loan).

  A conversational program can perform two types of program switching:

  Deferred program switch
  : Responds to the originating terminal but causes the next input from the terminal to go to another conversational program.

  Immediate program switch
  : Passes the conversation directly to another conversational program. The program passes the SPA (and, optionally, a message) to another conversational program without responding to the terminal. In this case, it is the next program's responsibility to respond to the originating terminal.

  A conversational program must:

  1. Retrieve the SPA and the message using GU and GN calls.

     If your MPP is starting this conversation, test the variable area of the SPA for zeros to determine if this is the beginning of the conversation. If the SPA does not contain zeros, it means that you started the conversation earlier and that you are now at a later stage in the conversation. If this is true, you would branch to the part of your program that processes this stage of the conversation to continue the conversation.

     If another MPP has passed control to your MPP to continue the conversation, the SPA contains the data you need to process the message, so you do not have to test it for zeros. Start processing the message immediately.
  2. Process the message, including handling any necessary database access.
  3. Send the output message to the terminal by using an ISRT call against the I/O PCB. This step can follow step 4.
  4. Store the data (that your program, or the program that you pass control to, needs to continue processing) in the SPA using an ISRT call to the I/O PCB. (This step can precede step 3.) IMS TM determines which segment is the SPA by examining the ZZZZ field of the segment shown in the tables below.

To end the conversation, move blanks to the area of the SPA that contains the transaction code, and then insert the SPA back to IMS TM by issuing an ISRT call and referencing the I/O PCB.

If your MPP passes the conversation to another conversational program, the steps after the program processes the message are somewhat different.

Also, your program should be designed to handle the situation that occurs when the first GU call to the I/O PCB does not return a message to the application program. This can happen if the person at the terminal cancels the conversation by entering the /EXIT command before the program issues a GU call. (This happens if the message from this terminal was the only message in the message queue for the program.)

The contents of SPA

The SPA that IMS TM gives your program when you issue a GU contains the four parts shown in the following tables.

Table 82. SPA format for AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI interfaces

Field Name	Field Length
LL	2
ZZZZ	4
TRANCODE	8
User Work Area	Variable

Table 83. SPA format for the PLITDLI interface

Field Name	Field Length
LLLL	4
ZZZZ	4
TRANCODE	8
User Work Area	Variable

The SPA format fields are:

LL or LLLL
A length field that gives the total length of the SPA. This length includes 2 bytes for the LL field. (For the PLITDLI interface, use a 4-byte field. Its contents include 4 bytes for LLLL, minus 2.)

ZZZZ
A 4-byte field reserved for IMS TM that your program must not modify.

TRANCODE
The 8-byte transaction code for this conversation.

User Work Area
A work area that you use to save the information that you need to continue the conversation. The length of this area depends on the length of the data you want to save. This length is defined at system definition.

When your program retrieves the SPA with a GU to start the conversation, IMS TM removes the transaction code from the message. In your first message segment, you receive only the data from the message that the person at the terminal entered.

The following list indicates the ways that an application program processes the SPA. The program must:

- Not modify the first 6 bytes of the SPA (LL and ZZZZ). IMS TM uses these fields to identify the SPA.

  If the program modifies the SPA, the program must return the SPA to IMS TM (or, for a program switch, to the other program).
- Not return the SPA to IMS TM more than once during one cycle of the conversation.
- Not insert the SPA to an alternate PCB that represents a nonconversational transaction code or a logical terminal. The program can use an alternate response PCB if it represents that same physical terminal as the originating logical terminal.

Restriction: If you are using MFS, the IMS TM does not always remove the transaction code.

The appearance of messages in a conversation

Because the first segment contains the SPA, conversational input messages are made up of at least two segments. The input message starts in the second message segment.

The input message segment in a conversation contains only the data from the terminal. During the first step in the conversation, IMS TM removes the transaction code from the input message and places it in the SPA. When the program issues the first `GU`, IMS TM returns the SPA. To retrieve the first message segment, the program must issue a `GN`.

The format for the output messages that you send to the terminal is no different than the format for output messages in nonconversational programs.

Saving information in the SPA

After you have processed the message and are ready to reply to the terminal, you can save the necessary data in the SPA. The part of the SPA in which you save data is the work area portion. Use the `ISRT` call to save data to the work area. This is a special use of the `ISRT` call, because you are not sending the SPA to a terminal, but rather saving it for future use.

If your program processes each stage of the conversation, you just issue an `ISRT` call to the I/O PCB and give the name of the I/O area that contains the SPA. For example:

```
ISRT   I/O PCB, I/O AREA
```

This returns the updated SPA to IMS TM so that IMS TM can pass it to your program at the next cycle of the conversation.

If you do not modify the SPA, you do not need to return it to IMS. However, the SPA will be passed by IMS TM to your program at the next cycle of the conversation.

Related concepts:

"Conversational processing using ROLB, ROLL, and ROLS" on page 422

"Passing the conversation to another conversational program" on page 423

Replying to the terminal

For a conversation to continue, the originating terminal must receive a response to each of its input messages. The person at the terminal cannot enter any more data to be processed (except IMS TM commands) until the response has been received at the terminal.

To continue the conversation, the program must respond to the originating terminal by issuing the required ISRT calls to send the output message to the terminal. To send a message to the originating terminal, the ISRT calls must reference either the TP PCB or an alternate response PCB. Use an alternate response PCB in a conversation when the terminal you are responding to has two components—for example, a printer and a punch—and you want to send the output message to a component that is separate from the component that sent the input message. If the program references an alternate response PCB, the PCB must be defined for the same physical terminal as the logical terminal that sent the input message.

The program can send only one output message to the terminal for each input message. Output messages can contain multiple segments, but the program cannot use the PURG call to send multiple output messages. If a conversational program issues a PURG call, IMS TM returns an AZ status code to the application program and does not process the call.

Conversational processing using ROLB, ROLL, and ROLS

Issuing a ROLB or ROLS in a conversational program causes IMS TM to back out the messages that the application program has sent.

If the application program issues a ROLB or ROLS and then reaches a commit point without sending the required response to the originating terminal, IMS TM terminates the conversation and sends the message DFS2171I NO RESPONSE CONVERSATION TERMINATED to the originating terminal.

If you issue ROLL during a conversation, IMS TM backs out the updates and cancels output messages, but it also terminates the conversation.

Conversational processing for modified message-driven IMS applications

The following processing considerations apply to modified message-driven IMS applications issuing the IMS ROLB call that can receive protected input messages from OTMA or APPC/MVS and issue outbound protected work to other z/OS Resource Recovery Services (RRS) resource managers:

- If a modified message-driven IMS application program with protected input issues a ROLB call, the ROLB call is isolated to the IMS application without affecting the entire protected unit of work. After the ROLB call is issued, the protected input message remains in process for the IMS application until a commit point is reached.
- If a modified message-driven IMS application program issues an outbound protected conversation, the outbound protected conversation is not included in the ROLB processing (that is, the outbound protected conversation is not backed out as part of the ROLB call). The modified message-driven IMS application program is responsible for explicitly cleaning up any outbound protected work to be backed out.

Related concepts:

"Conversational structure" on page 418

Passing the conversation to another conversational program

A conversational program can pass the conversation to another conversational program in by performing a deferred switch or a immediate switch.

A conversational program can pass the conversation to another conversational program in two ways:

- A deferred switch.

  The program can respond to the terminal but cause the next input from the terminal to go to another conversational program by:

  – Issuing an `ISRT` call against the I/O PCB to respond to the terminal
  – Placing the transaction code for the new conversational program in the SPA
  – Issuing an `ISRT` call referencing the I/O PCB and the SPA to return the SPA to IMS TM

  IMS TM then routes the next input message from the terminal to the program associated with the transaction code that was specified in the SPA. Other conversational programs can continue to make program switches by changing the transaction code in the SPA.

- An immediate switch.

  The program can pass the conversation directly to another conversational program by issuing an `ISRT` call against the alternate PCB that has its destination set to the other conversational program.

  The first `ISRT` call must send the SPA to the other program, but the program passing control can issue subsequent `ISRT` calls to send a message to the new program. If the program does this, in addition to routing the SPA to the other conversational program, IMS TM updates the SPA as if the program had returned the SPA to IMS. If the program does an immediate switch, the program cannot also return the SPA to IMS TM or respond to the original terminal.

Restrictions on passing the conversation

These are restrictions that apply to passing the conversation to another conversational program:

- When an immediate program switch occurs and the MPP receives an XE status code, the program attempts to insert the SPA to an alternate express PCB. Remove the EXPRESS=YES option from the PCB or define and use another PCB that is not express. This restriction prevents the second transaction from continuing the conversation if the first transaction abends after inserting the SPA.

  The person at the terminal can issue the `/SET CONV XX` command, where XX is the program that is to be scheduled in order to process the next step of the conversation.

- APPC or OTMA protected transactions do not allow immediate program or deferred program switches. If either of these switches occur, the MPP receives an X6 status code.

Defining the SPA size

Define the SPA size with the TRANSACT macro. An option to capture truncated data is also defined with the TRANSACT macro. The format is:

```
TRANSACT SPA=(size,STRUNC|RTRUNC)
```

The default is to support truncated data (STRUNC). When a conversation is initially started, and on each program switch, the truncated data option is checked and set or reset as specified. When the truncated data option is set, it remains set for the life of the conversation, or until a program switch occurs to a transaction that specifies that the option be reset.

For example, assume you have three transactions defined as follows:

TRANA SPA=100

TRANB SPA=050

TRANC SPA=150

For TRANC to receive the truncated data (which is the second 50 bytes from TRANA that TRANB does not receive) from TRANA, one of the following sets of specifications can be used:

- TRANA - STRUNC or none, TRANB - STRUNC or none, TRANC - STRUNC or none
- TRANA - RTRUNC, TRANB - STRUNC, TRANC - STRUNC or none

Conversational processing and MSC

If your installation has two or more IMS TM systems, and they are linked to each other through MSC, a program in one system can process a conversation that originated in another system.

- If a conversational program in system A issues an `ISRT` call that references a response alternate PCB in system B, system B does the necessary verification. This is because the destination is implicit in the input system. The verification that system B does includes determining whether the logical terminal that is represented by the response alternate PCB is assigned to the same physical terminal as the logical terminal that sent the input message. If it is not, system B (the originating system) terminates the conversation abnormally without issuing a status code to the application program.
- Suppose program A processes a conversation that originates from a terminal in system B. Program A passes the conversation to another conversational program by changing the transaction code in the SPA. If the transaction code that program A supplies is invalid, system B (the originating system) terminates the conversation abnormally without returning a status code to the application program.

Ending the conversation

To end the conversation, a program blanks out the transaction code in the SPA and returns it to IMS TM by issuing an `ISRT` call and referencing the I/O PCB and the SPA. This terminates the conversation as soon as the terminal has received the response.

The program can also end the conversation by placing a nonconversational transaction code in the transaction field of the SPA and returning the SPA to IMS. This causes the conversation to remain active until the person at the terminal has entered the next message. The transaction code will be inserted from the SPA into the first segment of the input message. IMS TM then routes this message from the terminal to the MPP or BMP that processes the transaction code that was specified in the SPA.

In addition to being ended by the program, a conversation can be ended by the person at the originating terminal, the master terminal operator, and IMS.

- The person at the originating terminal can end the conversation by issuing one of several commands:

  /EXIT The person at the terminal can enter the /EXIT command by itself, or the /EXIT command followed by the conversational identification number assigned by the IMS TM system.

  /HOLD
  The /HOLD command stops the conversation temporarily to allow the person at the terminal to enter other transactions while IMS TM holds the conversation. When IMS TM responds to the /HOLD command, it supplies an identifier that the person at the terminal can later use to reactivate the conversation. The /RELEASE command followed by this identifier reactivates the conversation.
- **/START LINE**. The master terminal operator can end the conversation by entering a /START LINE command (without specifying a PTERM) or /START NODE command for the terminal in the conversation or a /START USER command for a signed-off dynamic user in conversation.
- IMS TM ends a conversation if, after the program successfully issues a GU call or an ISRT call to return the SPA, the program does not send a response to the terminal. In this situation, IMS TM sends the message DFS2171I NO RESPONSE, CONVERSATION TERMINATED to the terminal. IMS TM then terminates the conversation and performs commit point processing for the application program.

Related concepts:

"Sending messages to other IMS application programs" on page 411

"Conversational structure" on page 418

Related tasks:

"Deferred program switching for conversational JMP applications" on page 664

"Immediate program switching for JMP and JBP applications" on page 662

Message switching in APPC conversations

With the system service DFSAPPC, you can transfer messages between separate LU 6.2 devices and between an LU 6.2 device and another terminal supported by IMS TM. Message delivery with DFSAPPC is asynchronous, so messages are held on the IMS TM message queue until they can be delivered.

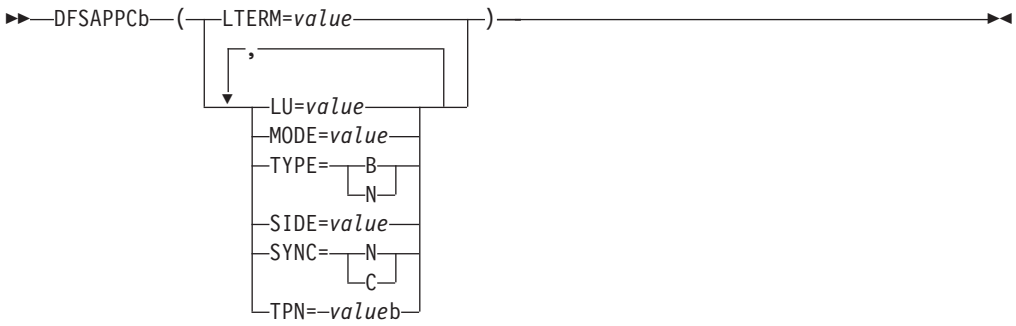
To send a message with DFSAPPC, specify the logical terminal name of an IMS TM terminal or the Transaction Program (TP) name of an LU 6.2 device.

DFSAPPC format

The message format for DFSAPPC is as follows:

```
DFSAPPC (options)user_data
```

DFSAPPC can be coded as follows:



A blank (␣) is required between DFSAPPC and the specified options.

Blanks are valid within the specified options except within keywords or values. Either commas or blanks can be used as delimiters between options, but because the use of commas is valid, the TP name must be followed by at least one blank.

If an LU 6.2 conversation has not been established from other sources (for example, during a CPI-C driven application program), DFSAPPC is used to establish the conversation with a partner LU 6.2 device. If no options are specified with DFSAPPC, IMS TM default options are used.

Option keywords

LTERM=
Specifies the LTERM name of an IMS TM logical terminal. An LTERM name can contain up to eight alphanumeric or national (@, $, #) characters. If you specify LTERM, you cannot specify the other option keywords.

LU=
Specifies the LU name of the partner in an LU 6.2 conversation. The LU name can contain up to eight alphanumeric or national characters, but the first character must be a letter or a national character. If both LU and SIDE options are specified, LU overrides the LU name contained in the side information entry but does not change that LU name.

If the LU name is a network-qualified name, it can be up to 17 characters long and consist of the network ID of the originating system, followed by a '.', and the LU name (for example, netwrkid.luname). The LU name and the network ID can be up to eight characters long.

MODE=
Specifies the MODE name of the partner in an LU 6.2 conversation. The MODE name can contain up to eight alphanumeric or national characters, but the first character must be a letter or a national character. If both MODE and SIDE option keywords are specified, MODE overrides the MODE name contained in the side information entry but does not change that MODE name.

TPN=
Specifies the transaction program (TP) name of the partner in an LU 6.2 conversation. The TP name can contain up to 64 characters from the 00640 character set. Because the character set allows commas, at least one blank must follow the TP name. If both TPN and SIDE option keywords are specified, TPN overrides the TP name contained in the side information entry but does not change that name.

Related Reading: The *CPI Communications Specification* describes the 00640 character set, which contains all alphanumeric and national characters and 20 special characters.

SIDE=
Specifies the name of the side information entry for the partner in an LU 6.2 conversation. The side information entry name can contain up to eight characters from the 01134 character set. If the SIDE option keyword is specified, it can be overridden with LU, MODE, and TPN option keywords.

Related Reading: The *CPI Communications Specification* describes the 01134 character set, which contains the uppercase alphabet and the digits, 0-9.

SYNC=N|C
Specifies the synchronization level of the LU 6.2 conversation. N selects none as the synchronization level, and C selects confirm as the synchronization level.

TYPE=B|M
Specifies the conversation type for the LU 6.2 conversation. B selects a basic conversation type, and M selects a mapped conversation type.

Processing conversations with APPC

APPC/IMS supports standard, modified, and CPI Communications driven application programs.

The three types of application programs supported by APPC/IMS

- Standard: No explicit use of CPI Communications facilities.
- Modified: Uses the I/O PCB to communicate with the original input terminal. Uses CPI Communications calls to allocate new conversations and to send and receive data.
- CPI Communications driven: Uses CPI Communications calls to receive the incoming message and to send a reply on the same conversation. Uses the DL/I APSB call to allocate a PSB to access IMS databases and alternate PCBs.

In the modified or CPI Communications driven application programs, if an APPC conversation is allocated with SYNCLVL=SYNCPT, z/OS manages the sync-point process for the APPC conversation participants: the application program and IMS. Transaction rollback and rescheduling is possible, because IMS issues the `SRRCMIT` or `SRRBACK` calls on behalf of the modified IMS APPC application program. If the CPI-C driven program is linked with the IMS stub code (DFSCPIR0) as required in previous releases, IMS also issues the SRRCMIT or SRRBACK calls. If the program is not linked with the stub code, then IMS is driven by the z/OS sync point manager when the application issues these calls. With z/OS as the sync point manager, failures can also be backed out.

You can schedule your standard and modified application programs locally and remotely using MSC or APPC/MVS. The logic flow for local scheduling differs from the logic flow for remote scheduling.

Scheduling programs remotely through MSC is not supported if an APPC/MVS conversation with SYNCLVL=SYNCPT is specified.

Ending the APPC conversation

You can end a conversation using LU 6.2 devices by issuing the CPI-C verb, DEALLOCATE, or by inserting a blank transaction code into the SPA for IMS conversational transactions.

Restriction: You cannot use the /EXIT command for LU 6.2 conversations.

Several error conditions can exist at the end of an LU 6.2 conversation:

- If your application program sends data to the LU 6.2 device just before deallocating conversation, IMS TM issues a SENDERROR and SENDDATA of the DFS1966 error message. This indicates that the transaction ended, but that the last message could not be delivered. For SENDERROR to be activated, specify a synchronization level of CONFIRM.
- If IMS TM encounters an error sending output from an IMS TM conversational transaction to the LU 6.2 device, the output is discarded, and the conversation is terminated for both IMS TM and LU 6.2.
- If an IMS TM conversational application program abends during an LU 6.2 conversation, a DFS555 error message is sent to the originating LU 6.2 device, and the conversation is terminated for both IMS TM and LU 6.2.

Coding a conversational program

Before coding a conversational program, you need to obtain the following information.

- The transaction code to use for a program to which you pass control
- The data that you should save in the SPA
- The maximum length of that data

A SPA contains four fields:

- The 2-byte length field.
- The 4-byte field that is reserved for IMS TM.
- The 8-byte transaction code.
- The work area where you store the conversation data. The length of this field is defined at system definition.

Standard IMS application programs

Standard IMS application programs use the existing IMS call interface. Application programs that use the IMS standard API can take advantage of the LU 6.2 protocols.

Standard IMS application programs use a DL/I GU call to get the incoming transaction. These standard IMS application programs also use DL/I ISRT calls to generate output messages to the same or different terminals, regardless of whether LU 6.2 is used. The identical program can work correctly for both LU 6.2 and non-LU 6.2 terminal types. IMS generates the appropriate calls to APPC/MVS services.

A non-message-driven BMP is considered a standard IMS application program when it does not use the explicit API.

When an advanced program-to-program communication (APPC) application program enters an IMS transaction that executes on a remote IMS, an LU 6.2 conversation is established between the APPC application program and the local IMS system. The local IMS is considered the partner LU of the LU 6.2 conversation. The transaction is then queued on the remote transaction queue of the local IMS system. From this point on, the transaction goes through normal

MSC processing. After the remote IMS system executes the transaction, the output is returned to the local IMS system and is then delivered to the originating LU 6.2 application program.

Modified IMS application programs

Modified IMS application programs use a DL/I `GU` call to get the incoming transaction. These modified IMS application programs also use DL/I `ISRT` calls to generate output messages to the same or different terminals, regardless of whether LU 6.2 is used.

A non-message-driven BMP is considered a modified standard IMS application program when it uses the explicit API. Unlike standard IMS application programs, modified IMS application programs use CPI Communications calls to allocate new conversations, and to send and receive data. IMS has no direct control of these CPI Communications conversations.

Modified IMS transactions are indistinguishable from standard IMS transactions until program execution. In fact, the same application program can be a standard IMS application on one execution, and a modified IMS application on a different execution. The distinction is simply whether the application program uses CPI Communications resources.

Modified IMS programs are scheduled by IMS TM, and the DL/I calls are processed by the DL/I language interface. The conversation, however, is maintained by APPC/MVS, and any failures that involve APPC/MVS are not backed out by IMS TM. The general format of a modified IMS application program is shown in the following code example.

```
• GU     IOPCB
     ALLOCATE
     SEND
     RECEIVE
     DEALLOCATE
• ISRT   IOPCB
```

Figure 79. General format of a modified DL/I application program

Restriction: The APPC conversation cannot span sync points. If the conversation is not deallocated before a sync point is reached, IMS causes the conversation to be terminated by issuing a clean TP call (ATBCMTP). A new APPC conversation can be allocated after each sync point.

When an APPC program enters an IMS transaction that executes on a remote IMS system, an LU 6.2 conversation is established between the APPC program and the local IMS system. The local IMS system is considered the partner LU of the LU 6.2 conversation. The transaction is then queued on the local IMS system's remote transaction queue. From this point on, the transaction goes through normal MSC processing. After the remote IMS system executes the transaction, the output is returned to the local IMS and is then delivered to the originating LU 6.2 program.

Related Reading: For more information on failure recovery and modified DL/I application program design, see *IMS Version 11 Application Programming APIs*.

CPI-C driven application programs

CPI Communications driven application programs are defined only in the APPC/MVS TP_Profile data set; they are not defined to IMS. Their definition is

dynamically built by IMS when a transaction is presented for scheduling by APPC/MVS, based on the APPC/MVS TP_Profile definition after IMS restart. The definition is keyed by TP name. APPC/MVS manages the TP_Profile information.

When a CPI Communications driven transaction program requests a PSB, the PSB must already be defined to IMS through the APPLCTN macro for system definition and through PSBGEN or ACBGEN when APPLCTN PSB= is specified. When APPLCTN GPSB= is specified, a PSBGEN or ACBGEN is not required.

CPI-C driven application programs must begin with the CPI-C verbs, ACCEPT and RECEIVE, to initiate the LU 6.2 conversation. You can then issue the APSB call to allocate a PSB for use by the application program. After the APSB call is issued, you can issue additional DL/I calls using the PCBs that were allocated. You then issue the SRRCMIT verb to commit changes or the SRRBACK verb to back out changes. To use SRRCMIT and SRRBACK, your application program must be linked with DFSCPIR0.

Restriction: The I/O PCB cannot be used for message processing calls by CPI-C driven application programs. See the description of each call for specific CPI restrictions.

To deallocate the PSB in use, issue the DPSB call. You can then issue another APSB call, or use the CPI-C verb, DEALLOCATE, to end the conversation.

CPI-C driven application programs are considered discardable (unless they are allocated with a SYNCLVL=SYNCPT) by IMS TM and are therefore not recovered automatically at system failure. If they are allocated with a SYNCLVL=SYNCPT, a two-phase commit process is used to recover from any failures. The general format of a CPI-C driven application program is shown in the following code example.

```
• ACCEPT
• RECEIVE
  – APSB
        GU DBPCB
        REPL DBPCB
        SRRCMIT
  – DPSB
• DEALLOCATE
```

Figure 80. General format of a CPI-C driven application program

Related concepts:

CPI-C driven application programs (Communications and Connections)

Processing conversations with OTMA

You can run IMS conversational transactions through OTMA.

Refer to *IMS Version 11 Communications and Connections*.

Backing out to a prior commit point: ROLL, ROLB, and ROLS calls

When a program determines that some of its processing is invalid, you can use these calls to remove the effects of its incorrect processing: Roll Back calls ROLL, ROLS using a database PCB, ROLS with no I/O area or token, and ROLB.

When you issue one of these calls, IMS does the following:

- Backs out the database updates that the program has made since the program's most recent commit point.
- Cancels the non-express output messages that the program has created since the program's most recent commit point.

The main difference among these calls is that ROLB returns control to the application program after backing out updates and canceling output messages, ROLS does not return control to the application program, and ROLL terminates the program with a user abend code of 0778. ROLB can return to the program the first message segment since the most recent commit point, but ROLL and ROLS cannot.

The ROLL and ROLB calls, and the ROLS call without a token specified, are valid when the PSB contains PCBs for Generalized Sequential Access Method (GSAM) data sets. However, segments inserted in the GSAM data sets since the last commit point are not backed out by these calls. An extended checkpoint-restart can be used to reposition the GSAM data sets when restarting.

You can use a ROLS call either to back out to the prior commit point or to back out to an intermediate backout point established by a prior SETS call. This section refers only to the form of ROLS that backs out to the prior commit point.

Related concepts:

"Backing out to an intermediate backout point: SETS, SETU, and ROLS" on page 288

Comparison of ROLB, ROLL, and ROLS

The following table provides a comparison of the ROLB, ROLL, and ROLS calls.

Table 84. Comparison of ROLB, ROLL, and ROLS.

Actions taken	ROLB	ROLL	ROLS
Back out database updates since the last commit point.	X	X	X
Cancel output messages created since the last commit point.	X[1]	X[1]	X[1]
Delete the message in process from the queue. Previous messages (if any) processed since the last commit point are returned to the queue to be reprocessed.		X	
Return the first segment of the first input message since the most recent commit point.	X[2]		
3303 abnormal termination and returns the processed input messages to the message queue.			X[3]
778 abnormal termination, no dump.		X	
No abend; program continues processing.	X		

Notes:

1. ROLB, ROLL, or ROLS cancel output messages sent with an express PCB unless the program issued a PURG.

   For example, if the program issues the following call sequence, MSG1 would be sent to its destination because the PURG tells IMS that MSG1 is complete and the I/O area now contains the first segment of the next message (which in this example is MSG2). MSG2, however, would be canceled:

   ```
   ISRT    EXPRESS PCB, MSG1
   PURG    EXPRESS PCB, MSG2
   ROLB    I/O PCB
   ```

   Because IMS has the complete message (MSG1) and because an express PCB is being used, the message can be sent before a commit point.
2. Returned only if you supply the address of an I/O area as one of the call parameters.
3. The transaction is suspended and requeued for subsequent processing.

ROLL

A ROLL call backs out the database updates and cancels any non-express output messages the program has created since the last commit point. It also deletes the current input message. Any other input messages processed since the last commit point are returned to the queue to be reprocessed. IMS then terminates the program with a user abend code 0778. This type of abnormal termination terminates the program without a storage dump.

When you issue a ROLL call, the only parameter you supply is the call function, ROLL.

You can use the ROLL call in a batch program. If your system log is on direct access storage, and if dynamic backout has been specified through the use of the BKO execution parameter, database changes since the last commit point will be backed out. Otherwise they will not be backed out. One reason for issuing ROLL in a batch program is for compatibility.

After backout is complete, the original transaction is discarded if it is discardable, and it is not re-executed. IMS issues the APPC/MVS verb ATBCMTP TYPE(ABEND) specifying the TPI to notify remote transaction programs. Issuing the APPC/MVS verb causes all active conversations (including any spawned by the application program) to be DEALLOCATED TYP(ABEND_SVC).

ROLB

The advantage of using ROLB is that IMS returns control to the program after executing ROLB, so the program can continue processing.

The parameters for ROL are:

- The call function ROLB
- The name of the I/O PCB or AIB

The total effect of the ROLB call depends on the type of IMS application that issued it.

- For current IMS application programs:

After IMS backout is complete, the original transaction is represented to the IMS application program. Any resources that cannot be rolled back by IMS are ignored. For example, output sent to an express alternate PCB and a PURG call is issued before the ROLB.

- For modified IMS application programs:

  The same consideration for the current IMS application programs applies. It is the responsibility of the application program to notify any spawned conversations that a ROLB was issued.

- For CPI-C driven IMS application programs:

  Only IMS resources are affected. All database changes are backed out. Any messages inserted to nonexpress alternate PCBs are discarded. Also, any messages inserted to express PCBs that have not had a PURGE call are discarded. It is the responsibility of the application program to notify the originating remote program and any spawned conversations that a ROLB call was issued.

In MPPs and transaction-oriented BMPs

If the program supplies the address of an I/O area as one of the ROLB parameters, the ROLB call acts as a message retrieval call and returns the first segment of the first input message since the most recent commit point. This is true only if the program has issued a GU call to the message queue since the last commit point; it if has not, it was not processing a message when it issued the ROLB call.

If the program issues a GN to the message queue after issuing the ROLB, IMS returns the next segment of the message that was being processed when ROLB was issued. If there are no more segments for that message, IMS returns a QD status code.

If the program issues a GU to the message queue after the ROLB call, IMS returns the first segment of the next message to the application program. If there are no more messages on the message queue for the program to process, IMS returns a QC status code to the program.

If you include the I/O area parameter, but you have not issued a successful GU call to the message queue since the last commit point, IMS returns a QE status code to your program.

If you do not include the address of an I/O area in the ROLB call, IMS does the same things for you. If the program has issued a successful GU in the commit travel, and then issues a GN, IMS returns a QD status code. If the program issues a GU after the ROLB, IMS returns the first segment of the next message, or a QC status code if there are no more messages for the program.

If you have not issued a successful GU since the last commit point, and you do not include an I/O area parameter on the ROLB call, IMS backs out the database updates and cancels the output messages created since the last commit point.

In batch programs

If your system log is on direct access storage, and if dynamic backout has been specified through the use of the BKO execution parameter, you can use the ROLB call in a batch program. The ROLB call does not process messages as it does for message processing programs (MPPs); it backs out the database updates since the last commit point and returns control to your program. You cannot specify the address of an I/O area as one of the parameters on the call; if you do, an AD

status code is returned to your program. You must, however, have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB.

Related Reading: For more information on using the CMPAT keyword, see *IMS Version 11 System Utilities*. For information on coding the ROLB call, see the topic "ROLB Call" in *IMS Version 11 Application Programming APIs*.

ROLS

You can use the ROLS call to back out to the prior commit point and return the processed input messages to IMS for later reprocessing.

In your program, you can either:

- Issue the ROLS call using the I/O PCB but without an I/O area or token in the call. The parameters for this form of the ROLS call are:
  - The call function ROLS
  - The name of the I/O PCB or AIB
- Issue the ROLS call using a database PCB that has received one of the data-unavailable status codes. This has the same result as if unavailable data were encountered, and the INIT call was not issued. ROLS must be the next call for that PCB. Intervening calls using other PCBs are permitted.

On a ROLS with a token, message queue repositioning can occur for all non-express messages including all messages processed by IMS. This processing using APPC/MVS calls and includes the initial message segments. The original input transaction can be represented to the IMS application program. Input and output positioning is determined by the SETS call. This positioning applies to current and modified IMS application programs but does not apply to CPI-C driven IMS programs. The IMS application program must notify all remote transaction programs of the ROLS.

On a ROLS without a token, IMS issues the APPC/MVS verb, ATBCMTP TYPE(ABEND), specifying the TPI. Issuing this verb causes all conversations associated with the application program to be DEALLOCATED TYPE(ABEND_SVC). If the original transaction was entered from an LU 6.2 device and IMS received the message from APPC/MVS, a discardable transaction is discarded rather than being placed on the suspend queue like a non-discardable transaction.

Related Reading: For more information on LU 6.2, see *IMS Version 11 Communications and Connections*.

The parameters for this form of the ROLS call are:

- The call function, ROLS
- The name of the DB PCB that received the BA or BB status code

In both of the ways to use ROLS calls, the ROLS call causes a 3303 abnormal termination and does not return control to the application program. IMS keeps the input message for future processing.

Backing out to an intermediate backout point: SETS/SETU and ROLS

You can use a ROLS call either to back out to an intermediate backout point established by a prior SETS or SETU call or to back out to the prior commit point.

This section refers only to the form of ROLS that backs out to the intermediate backout point. For information about the other form of ROLS, see 'Backing out to a prior commit point: ROLL, ROLB, and ROLS calls'.

The ROLS call that backs out to an intermediate point backs out only DL/I changes. This version of the ROLS call does not affect CICS changes using CICS file control or CICS transient data.

The SETS and ROLS calls set intermediate backout points within the call processing of the application program and then backout database changes to any of these points. Up to nine intermediate backout points can be set. The SETS call specifies a token for each point. IMS then associates this token with the current processing point. A subsequent ROLS call, using the same token, backs out all database changes and discards all non-express messages that were performed following the SETS call with the same token. The figure below shows how the SETS and ROLS calls work together.

In addition, to assist the application program in reestablishing other variables following a ROLS call, user data can be included in the I/O area of the SETS call. This data is then returned when the ROLS call with the same token is issued.

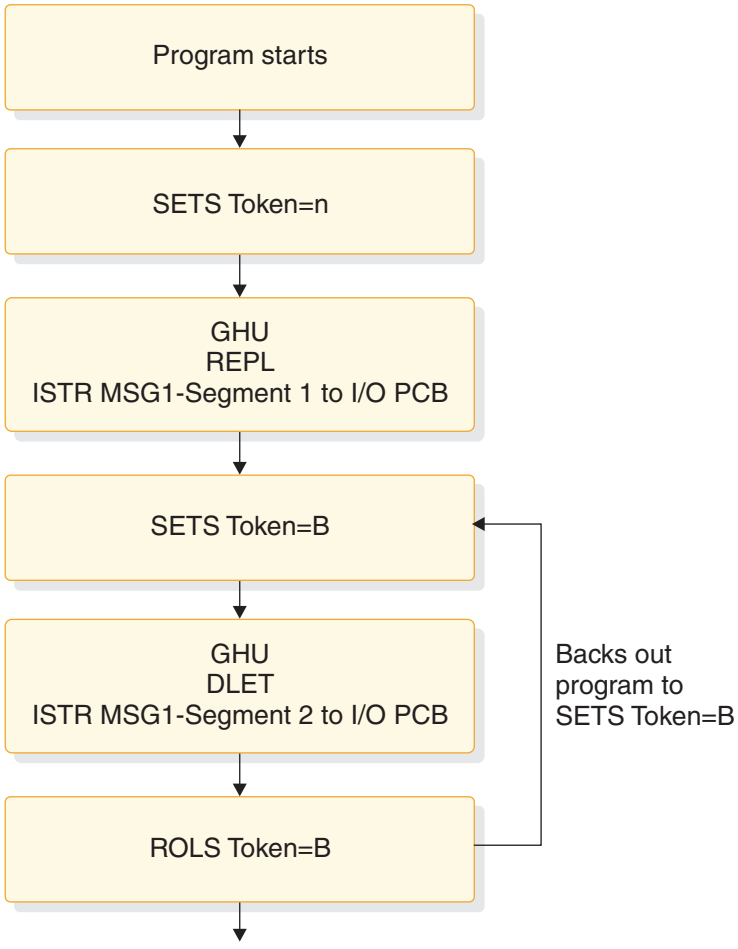


Figure 81. SETS and ROLS calls working together

SETS/SETU

The SETS call sets up to nine intermediate backout points or cancels all existing backout points. By using the SETS call, you can back out pieces of work. If the necessary data to complete one piece of work is unavailable, you can complete a different piece of work and then return to the former piece.

To set an intermediate backout point, issue the call using the I/O PCB and include an I/O area and a token. The I/O area has the format LLZZ user-data, where LL is the length of the data in the I/O area including the length of the LLZZ portion. The ZZ field must contain binary zeros. The data in the I/O area is returned to the application program on the related ROLS call. If you do not want to save some data to be returned on the ROLS call, you must set the LL that defines the length of the I/O area to 4.

For PLITDLI, you must define the LL field as a fullword rather than a halfword as it is for the other languages. The content of the LL field for PLITDLI is consistent with the I/O area for other calls using the LLZZ format; that is, the content is the total length of the area including the length of the 4-byte LL field minus 2.

A 4-byte token associated with the current processing point is also required. This token can be a new token for this program execution or match a token issued by a preceding SETS call. If the token is new, no preceding SETS calls are canceled. If the token matches the token of a preceding SETS call, the current SETS call assumes that position. In this case, all SETS calls that were issued subsequent to the SETS call with the matching token are canceled.

The parameters for this form of the SETS call are:

- The call function SETS
- The name of the I/O PCB or AIB
- The name of the I/O area containing the user data
- The name of an area containing the token

For the SETS call format, see the topic 'SETS/SETU Call' in *IMS Version 11 Application Programming APIs*.

To cancel all previous backout points, the call is issued using the I/O PCB but does not include an I/O area or a token. When no I/O area is included in the call, all intermediate backout points set by prior SETS calls are canceled.

The parameters for this form of the SETS call are:

- The call function SETS
- The name of the I/O PCB or AIB

Because it is not possible to back out committed data, commit point processing causes all outstanding SETS to be canceled.

If PCBs for DEDB, MSDB, and GSAM organizations are in the PSB, or if the program accesses an attached subsystem, a partial backout is not possible. In that case, the SETS call is rejected with an SC status code. If the SETU call is used instead, it is not rejected because of unsupported PCBs, but returns an SC status code as a warning that the PSB contains unsupported PCBs and the function is not applicable to these unsupported PCBs.

Related Reading: For the status codes that are returned after the SETS call and the explanation of those status codes and the response required, see *IMS Version 11 Application Programming APIs*.

ROLS

The ROLS call backs out database changes to a processing point set by a previous SETS or SETU call, or to the prior commit point and returns the processed input messages to the message queue.

To back out database changes and message activity that have occurred since a prior SETS call, you issue the ROLS call using the I/O PCB and specifying an I/O area and token in the call. If the token does not match a token set by a preceding SETS call, an error status is returned. If the token does match the token of a preceding SETS call, the database updates made since this corresponding SETS call are backed out, and all non-express messages inserted since the corresponding SETS are discarded. The ROLS call returns blanks if the call is processed, and returns a status code if an error or warning occurs. If you are using SETU with ROLS and have an external subsystem, the ROLS call will not be rejected, but an RC status code will be returned as a warning. All SETS points that were issued as part of the processing that was backed out are then canceled, and the existing database position for all supported PCBs is reset. For the ROLS call format, see the topic "ROLB Call" in *IMS Version 11 Application Programming APIs*.

The parameters for this form of the ROLS call are:

- The call function ROLS
- The name of the I/O PCB or AIB
- The name of the I/O area to receive the user data
- The name of an area containing the 4-byte token

Related reading: For the status codes that are returned after the ROLS call and the explanations of those status codes and the response required, see *IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes*.

Writing message-driven programs

A message-driven program is similar to an MPP: it retrieves messages and processes them, and it can read and update MSDBs, DEDBs, and full-function databases.

Message-driven programs can send messages to these destinations:

- The logical terminal that sent the input message, by issuing an ISRT call referencing the I/O PCB
- A different component of the physical terminal that sent the input message, by issuing an ISRT call referencing an alternate response PCB
- A different physical terminal from the one that sent the input message, by issuing an ISRT call referencing an alternate PCB

The message processing functions available to a message-driven program have some restrictions. These restrictions apply only to messages received or sent by the I/O PCB. The input message for a message-driven program must be a single segment message. Therefore, GU is the only call you can use to obtain the input message. The response message sent by the I/O PCB also must be a single segment message.

The transactions are in the response mode. This means that you must respond before the next message can be sent. You cannot use SPAs because a message-driven program cannot be a conversational program.

Not all of the system service calls are available. These system service calls are valid in a message-driven region:

CHKP (basic)

DEQ

INIT

LOG

SETS

ROLB

ROLS

However, other conditions might restrict their function in this environment. The options or calls issued using alternate terminal PCBs have no constraints.

Coding DC calls and data areas

The way you code DC calls and data areas depends on the application programming language you use.

Before coding your program

In addition to the information you need about the database processing that your program does, you need to know about message processing. Before you start to code, be sure you are not missing any of this information. Also, be aware of the standards at your installation that affect your program.

Information you need about your program's design:

- The names of the logical terminals that your program will communicate with
- The transaction codes, if any, for the application program's MPP skeleton to which your program will send messages
- The DC call structure for your program
- The destination for each output message that you send
- The names of any alternate destinations to which your program sends messages

Information you need about input messages:

- The size and layout of the input messages your program will receive (if possible)
- The format in which your program will receive the input messages
- The editing routine your program uses
- The range of valid data in input messages
- The type of data that input messages will contain
- The maximum and minimum length of input message segments
- The number of segments in a message

Information you need about output messages:

- The format in which IMS expects to receive output from your application program MPP skeleton
- The destination for the output messages

- The maximum and minimum length of output message segments

MPP code examples

Your MPP application can be written in assembler language, COBOL, C, Pascal, and PL/I.

In the following code examples, the programs do not have all the processing logic that a typical MPP has. The purpose of providing these programs is to show you the basic MPP structure in assembler language, COBOL, C language, Pascal, and PL/I. All the programs follow these steps:

1. The program retrieves an input message segment from a terminal by issuing a `GU` call to the I/O PCB. This retrieves the first segment of the message. Unless this message contains only one segment, your program issues `GN` calls to the I/O PCB to retrieve the remaining segments of the message. IMS places the input message segment in the I/O area that you specify in the call. In each of skeleton MPP examples, this is the MSG-SEG-IO-AREA.
2. The program retrieves a segment from the database by issuing a `GU` call to the DB PCB. This call specifies an SSA, SSA-NAME, to qualify the request. IMS places the database segment in the I/O area specified in the call. In this case, the I/O area is called DB-SEG-IO-AREA.
3. The program sends an output message to an alternate destination by issuing an `ISRT` call to the alternate PCB. Before issuing the `ISRT` call, the program must build the output message segment in an I/O area, and then the program specifies the I/O area in the `ISRT` call. The I/O area for this call is ALT-MSG-SEG-OUT.

The sample program is simplified for demonstration purposes; for example, the call to initiate sync point is not shown in the sample program. Include other IMS calls in a complete application program.

Coding your MPP program in assembler language

The coding conventions of an assembler language MPP are the same as those for a DL/I assembler program.

An assembler language MPP receives a PCB parameter list address in register 1 when it executes its entry statement. The first address in this list is a pointer to the TP PCB; the addresses of any alternate PCBs that the program uses come after the I/O PCB address, and the addresses of the database PCBs that the program uses follow. Bit 0 of the last address parameter is set to 1.

Coding your MPP program in C language

The program shown below is a skeleton MPP written in C language.

The numbers to the right of the program refer to the notes that follow the program. All storage areas that are referenced in the parameter list of your C language application program call to IMS can reside in the extended virtual storage area.

Skeleton MPP written in C

```
                                                                NOTES
#pragma runopts(env(IMS),plist(IMS))                             1
  #include <ims.h>
  #include <stdio.h>
  /*                                                  */
  /*                    ENTRY POINT                   */
  /*                                                  */
```

```
 main() {                                                              2
   static const char func_GU[4] = "GU  ";                              3
   static const char func_ISRT[4] = "ISRT";
  .
   #define io_pcb ((IO_PCB_TYPE *)(_pcblist[0])
4
   #define alt_pcb (_pcblist[1])
   #define db_pcb  (_pcblist[2])
  .
   int rc;
5
  .
   #define io_pcb ((IO_PCB_TYPE *)(_pcblist[0])
6
   #define alt_pcb (_pcblist[1])
   #define db_pcb  (_pcblist[2])
  .
  rc = ctdli(func_GU, io_pcb, msg_seg_io_area);
7
 .
   rc = ctdli(func_GU, db_pcb, db_seg_io_area, ssa_name);
8
 .
   rc = ctdli(func_ISRT, alt_pcb, alt_msg_seg_out);
9
 .
 }
10
C language interface
11
```

Note:

1. The env(IMS) establishes the correct operating environment and the plist(IMS) establishes the correct parameter list, when invoked under IMS. The ims.h header file contains declarations for PCB layouts, __pcblist, and the ctdli routine. The PCB layouts define masks for the DB PCBs that the program uses as structures. These definitions make it possible for the program to check fields in the DB PCBs.

   The stdio.h header file contains declarations for sprintf, which is useful for building SSAs.
2. After IMS has loaded the application program's PSB, IMS passes control to the application program through this entry point.
3. These are convenient definitions for the function codes and could be in one of your include files.
4. These could be structures, with no loss of efficiency.
5. The return code (status value) from DL/I calls can be returned and used separately.
6. The C language run-time sets up the __pcblist values. The order in which you refer to the PCBs must be the same order in which they have been defined in the PSB: first the TP PCB, then any alternate PCBs that your program uses, and finally the database PCBs that your program uses.
7. The program issues a `GU` call to the I/O PCB to retrieve the first message segment. You can leave out the rc =, and check the status in some other way.
8. The program issues a `GU` call to the DB PCB to retrieve a database segment. The function codes for these two calls are identical; the way that IMS identifies them is by the PCB to which each call refers.
9. The program then sends an output message to an alternate destination by issuing an `ISRT` call to an alternate PCB.

10. When there are no more messages for the program to process, the program returns control to IMS by returning from main or by calling exit().
11. IMS provides a language interface module (DFSLI000) that gives a common interface to IMS. This module must be made available to the application program at bind time.

Coding your MPP program in COBOL

The program shown below is a skeleton MPP in COBOL that shows the main elements of an MPP.

The numbers to the right of each part of the program refer to the notes that follow the program. If you plan to preload your IBM COBOL for z/OS & VM program, you must use the compiler option RENT. Alternatively, if you plan to preload your VS COBOL II program, you must use the compiler options RES and RENT.

If you want to use the IBM COBOL for z/OS & VM compiler to compile a program that is to execute in AMODE(31) on z/OS, you must use the compiler option RENT. Alternatively, if you want to use the VS COBOL II compiler to compile a program that is to execute in AMODE(31) on z/OS, you must use the compiler options RES and RENT. All storage areas that are referenced in the parameter lists of your calls to IMS can optionally reside in the extended virtual storage area.

IBM COBOL for z/OS & VM and VS COBOL II programs can coexist in the same application.

Skeleton MPP written in COBOL

```
                                                                NOTES:
ENVIRONMENT DIVISION.
    .
    .
    .
DATA DIVISION.
WORKING-STORAGE SECTION.                                        1
    77 GU-CALL PICTURE XXXX VALUE 'GU  '.
    77 ISRT-CALL PICTURE XXXX VALUE 'ISRT'.
    77 CT PICTURE S9(5) COMPUTATIONAL VALUE +4.
    01 SSA-NAME.
    .
    01 MSG-SEG-IO-AREA.                                        2
    .
    01 DB-SEG-IO-AREA.
    .
    01 ALT-MSG-SEG-OUT.
    .
LINKAGE SECTION.
    01 IO-PCB.                                                 3
    .
    01 ALT-PCB.
    .
    01 DB-PCB.
    .
PROCEDURE DIVISION USING IO-PCB, ALT-PCB, DB-PCB              4
    .
    CALL 'CBLTDLI' USING GU-CALL, IO-PCB,                     5
        MSG-SEG-IO-AREA.
    .
    CALL 'CBLTDLI' USING GU-CALL, DB-PCB,                     6
        DB-SEG-IO-AREA, SSA-NAME.
    .
    CALL 'CBLTDLI' USING ISRT-CALL, ALT-PCB,                  7
```

```
          ALT-MSG-SEG-OUT.
       .
       GOBACK.                                          8
COBOL LANGUAGE INTERFACE                                9
```

Note:

1. To define each of the call functions that your program uses, use a 77 or 01 level working-storage statement. Assign the value to the call function in a picture clause defined as four alphanumeric characters.
2. Use a 01 level working-storage statement for each I/O area that you will use for message segments.
3. In the linkage section of the program, use a 01 level entry for each PCB that your program uses. You can list the PCBs in the order that you list them in the entry statement, but this is not a requirement.
4. On the procedure statement, list the PCBs that your program uses in the order they are defined in the program's PSB: first the TP PCB, then any alternate PCBs, and finally the database PCBs that your program uses.
5. The program issues a GU call to the I/O PCB to retrieve the first segment of an input message.
6. The program issues a GU call to the DB PCB to retrieve the segment that would be described in the SSA-NAME area.
7. The program sends an output message segment to an alternate destination by using an alternate PCB.
8. When no more messages are available for your MPP to process, you return control to IMS by issuing the GOBACK statement.

   If you compile all of your COBOL programs in the task with VS COBOL II, you can use the GOBACK statement with its normal COBOL-defined semantics.

   Attention: The STOP RUN and EXIT PROGRAM statements are not supported. Using these statements might cause unpredictable results or abends.
9. If the COBOL compiler option NODYNAM is specified, you must link edit the language interface module, DFSLI000, with your compiled COBOL application program. If the COBOL compiler option DYNAM is specified, do not link edit DFSLI000 with your compiled COBOL program.

Coding your MPP program in Pascal

The program shown below is a skeleton MPP written in Pascal.

The numbers to the right of the program refer to the notes that follow the program. All storage areas that are referenced in the parameter list of your Pascal application program's call to IMS can reside in the extended virtual storage area.

Skeleton MPP written in Pascal

```
                    NOTES:
segment PASCIMS;                                        1
type
  CHAR4 = packed array [1..4] of CHAR;2
  CHARn = packed array [1..n] of CHAR;
  IOPCBTYPE = record                                    3
                (* Field declarations *)
              end;
  ALTPCBTYPE = record
               (* Field declarations *)
             end;
  DBPCBTYPE = record
              (* Field declarations *)
            end;
```

```
procedure PASCIMS (var SAVE: INTEGER;                    4
                   var IOPCB:  IOPCBTYPE;
                   var ALTPCB: ALTPCBTYPE;
                   var DBPCB:  DBPCBTYPE); REENTRANT;
procedure PASCIMS;
type                                                     5
  SSATYPE   = record
                (* Field declarations *)
              end;

  MSG_SEG_IO_AREA_TYPE = record
                           (* Field declarations *)
                         end;

  DB_SEG_IO_AREA_TYPE = record
                          (* Field declarations *)
                        end;

  ALT_MSG_SEG_OUT_TYPE = record
                           (* Field declarations *)
                         end;
var                                                      6
  MSG_SEG_IO_AREA : MSG_SEG_IO_AREA_TYPE;
  DB_SEG_IO_AREA : DB_SEG_IO_AREA_TYPE;
  ALT_MSG_SEG_OUT : ALT_MSG_SEG_OUT_TYPE;
const                                                    7
  GU   = 'GU  ';
  ISRT = 'ISRT';
  SSANAME  = SSATYPE(...);
procedure PASTDLI; GENERIC;                              8
begin
  PASTDLI(const GU,                                      9
          var   IOPCB,
          var   MSG_SEG_IO_AREA);
  PASTDLI(const GU,                                      10
          var   DBPCB,
          var   DB_SEG_IO_AREA,
          const SSANAME);
  PASTDLI(const ISRT,                                    11
          var   ALTPCB,
          var   ALT_MSG_SEG_OUT);
end;                                                     12
Pascal language interface                                13
```

Note:

1. Define the name of the Pascal compile unit.
2. Define the data types needed for the PCBs used in your program.
3. Define the PCB data types used in your program.
4. Declare the procedure heading for the REENTRANT procedure called by IMS. The first word in the parameter list should be an INTEGER, which is reserved for VS Pascal's use, and the rest of the parameters will be the addresses of the PCBs received from IMS.
5. Define the data types needed for the SSAs and I/O areas.
6. Declare the variables used for the SSAs and I/O areas.
7. Define the constants (function codes, SSAs, and so forth) used in the PASTDLI DL/I calls.
8. Declare the IMS interface routine with the GENERIC Directive. GENERIC identifies external routines that allow multiple parameter list formats. A GENERIC routine's parameters are “declared” only when the routine is called.

9. The program issues a GU call to the I/O PCB to retrieve the first segment of an input message. The declaration of the parameters in your program might differ from this example.
10. The program can issue a GU call to a DB PCB to retrieve a database segment. The function codes for these two calls are identical; the way that IMS distinguishes between them is by the PCB to which each call refers. The declaration of the parameters in your program might differ from this example.
11. The program sends an output message segment to an alternate destination by issuing an ISRT call to an alternate PCB. The declaration of the parameters in your program might differ from this example.
12. When there are no more messages for your MPP to process, you return control to IMS by exiting the PASCIMS procedure. You can also code a RETURN statement to leave at another point.
13. You must bind your program to the IMS language interface module, DFSLI000, after you have compiled your program.

Coding your MPP program in PL/I

The following program is a skeleton MPP written in PL/I.

The numbers to the right of the program refer to the notes for the program. All storage areas that are referenced in the parameter list of your PL/I application program call to IMS can optionally reside in the extended virtual storage area.

If you plan to execute PL/I programs in 31-bit addressing mode, see Enterprise PL/I for z/OS Programming Guide.

Skeleton MPP written in PL/I

```
                                                                  NOTES
/*                                                              */
/*                     ENTRY POINT                              */
/*                                                              */
    UPDMAST:  PROCEDURE (IO_PTR, ALT_PTR, DB_PTR)                    1
    OPTIONS (MAIN);
DCL   FUNC_GU     CHAR(4)   INIT('GU  ');                            2
DCL   FUNC_ISRT   CHAR(4)   INIT('ISRT');
.
DCL   SSA_NAME...;
.
DCL   MSG_SEG_IO_AREA    CHAR(n);                                    3
DCL   DB_SEG_IO_AREA     CHAR(n);
DCL   ALT_MSG_SEG_OUT    CHAR(n);
.
DCL   1 IO_PCB BASED (IO_PTR),...;                                   4
DCL   1 ALT_PCB BASED (ALT_PTR),...;
DCL   1 DB_PCB BASED (DB_PTR),...;
.
DCL   THREE FIXED BINARY(31) INIT(3);                                5
DCL   FOUR FIXED BINARY(31)  INIT(4);
DCL   PLITDLI ENTRY EXTERNAL;
.
CALL PLITDLI (THREE, FUNC_GU, IO_PTR, MSG_SEG_IO_AREA);              6
.
CALL PLITDLI (FOUR, FUNC_GU, DB_PTR, DB_SEG_IO_AREA,                 7
     SSA_NAME);
.
CALL PLITDLI (THREE, FUNC_ISRT, ALT_PTR, ALT_MSG_SEG_OUT);           8
.
END UPDMAST;                                                         9
PL/I LANGUAGE INTERFACE                                              10
```

Note:

1. This is the standard entry point to a PL/I Optimizing Compiler MPP. This statement includes a pointer for each PCB that the MPP uses. You must refer to the PCBs in the same order as they are listed in the PSB: first the TP PCB, then any alternate PCBs that your program uses, and finally the database PCBs that your program uses.
2. The program defines each call function that it uses in its data area. In PL/I, you define the function codes as character strings and assign the appropriate values to them.
3. Define PCB Masks as major structures based on the addresses passed in the PROCEDURE statement. Although not shown in the example, you will code the appropriate additional fields in the structure, depending on the type of PCB to which the mask is associated.
4. To define your PCBs, use major structure declarations.
5. PL/I calls have a parameter that is not required in COBOL programs or assembler language programs. This is the parmcount, and it is always the first parameter. You define the values that your program will need for the parmcount in each of its calls. The parmcount gives the number of parameters that follow parmcount itself.
6. The program issues a `GU` call to the I/O PCB to retrieve the first message segment.
7. The program can issue a `GU` call to a DB PCB to retrieve a database segment. The function codes for these two calls are identical; the way that IMS distinguishes between them is by the PCB to which each call refers.
8. The program then sends an output message to an alternate destination by issuing an `ISRT` call to an alternate PCB.
9. When there are no more messages for the program to process, the program returns control to IMS by issuing the END statement or the RETURN statement.
10. You must bind your program to the IMS language interface module, DFSLI000, after you have compiled your program.

Message processing considerations for DB2

For the most part, the message processing function of a dependent region that accesses DB2 databases is similar to that of a dependent region that accesses only DL/I databases. The method each program uses to retrieve and send messages and back out database changes is the same.

The differences are:

- DL/I statements are coded differently from SQL (structured query language) statements.
- When an IMS TM application program receives control from IMS TM, IMS has already acquired the resources the program is able to access. IMS TM schedules the program, although some of the databases are not available. DB2 does not allocate resources for the program until the program issues its first SQL statement. If DB2 cannot allocate the resources your program needs, your program can optionally receive an initialization error when it issues its first SQL call.
- When an application issues a successful checkpoint call or a successful message `GU` call, DB2 closes any cursors that the program is using. This means that your program should issue its OPEN CURSOR statement after a checkpoint call or a message `GU`.

IMS TM and DB2 work together to keep data integrity in these ways:

- When your program reaches a commit point, IMS TM makes any changes that the program has made to DL/I databases permanent, releases output messages for their destinations, and notifies DB2 that the program has reached a commit point. DB2 then makes permanent any changes that the program has made to DB2 databases.
- When your program terminates abnormally or issues one of the IMS TM rollback calls (`ROLB`, `ROLS` without a token, or `ROLL`), IMS TM cancels any output messages your program has produced, backs out changes your program has made to DL/I databases since the last commit point, and notifies DB2. DB2 backs out the changes that the program has made to DB2 databases since the last commit point.

Through the Automated Operator Interface (AOI), IMS TM application programs can issue DB2 commands and IMS TM commands. To issue DB2 commands, the program issues the IMS TM `/SSR` command followed by the DB2 command. The output of the `/SSR` command is routed to the master terminal operator (MTO).

Chapter 25. IMS Spool API

The IMS Spool API support provides feedback to the application program when IMS detects errors in the print data set options of the CHNG and SETO calls.

For convenience, your application program can display these errors by sending a message to an IMS printer or by performing another action that lets you examine the parameter lists and feedback area without looking at a dump listing. This information applies only to the calls as they are used with Spool API support.

Managing the IMS Spool API overall design

The IMS Spool API (application programming interface) is an expansion of the IMS application program interface that allows applications to interface directly to JES and create print data sets on the job entry subsystem (JES) spool. These print data sets can then be made available to print managers and spool servers to serve the needs of the application.

IMS Spool API design

The IMS Spool API design provides the application program with the ability to create print data sets on the JES spool using the standard DL/I call interface.

The functions provided are:

Definition of the data set output characteristics

Allocation of the data set

Insertion of lines of print into the data set

Closing and deallocation of the data set

Backout of uncommitted data within the limits of the JES interface

Assistance in controlling an in-doubt print data set

The IMS Spool API support uses existing DL/I calls to provide data set allocation information and to place data into the print data set. These calls are:

- The CHNG call. This call is expanded so that print data set characteristics can be specified for the print data set that will be allocated. The process uses the alternate PCB as the interface block associated with the print data set.
- The ISRT call. This call is expanded to perform dynamic allocation of the print data set on the first insert, and to write data to the data set. The data set is considered in-doubt until the unit of work (UOW) terminates. If possible, the sync point process deletes all in-doubt data sets for abending units of work and closes and deallocates data sets for normally terminating units of work.
- The SETO call. This is a call, SETO (Set Options), introduced by this support. Use this call to create dynamic output text units to be used with the subsequent CHNG call. If the same output descriptor is used for many print data sets, the overhead can be reduced by using the SETO call to prebuild the text units necessary for the dynamic output process.

© Copyright IBM Corp. 1974, 2015

Sending data to the JES spool data sets

Application programs can send data to the JES spool data sets using the same method that is used to send output to an alternate terminal. Use the DL/I call to change the output destination to a JES spool data set.

Use the DL/I `ISRT` or `PURG` call to insert a message.

The options list parameter on the `CHNG` and `SETO` calls contains the data set printer processing options. These options direct the output to the appropriate IMS Spool API data set. These options are validated for the DL/I call by the MVSScheduler JCL Facility (SJF). If the options are invalid, error codes are returned to the application. To receive the error information, the application program specifies a feedback area in the `CHNG` or `SETO` DL/I call parameter list. If the feedback area is present, information about the options list error is returned directly to the application.

IMS Spool API performance considerations

The IMS Spool API interface uses z/OS services within an IMS application while minimizing the performance impact of the z/OS services on the other IMS transactions and services.

For this reason, the IMS Spool API support places the print data directly on the JES spool at insert time instead of using the IMS message queue for intermediate storage. The processing of IMS Spool API requests is performed under the TCB of the dependent region to ensure maximum usage of N-way processors. This design reduces the error recovery and JES job orientation problems.

JES initiator considerations

Because the dependent regions are normally long-running jobs, some of the initiator or job specifications might must be changed if the dependent region is using the IMS Spool API.

You might need to limit the amount of JES spool space used by the dependent region to contain the dynamic allocation and deallocation messages. For example, you can use the JOB statement MSGLEVEL to eliminate the dynamic allocation messages from the job log for the dependent region. You might be able to eliminate these messages for dependent regions executing as z/OS started tasks.

Another initiator consideration is the use of the JES job journal for the dependent region. If the job step has a journal associated with it, the information for z/OS checkpoint restart is recorded in the journal. Because IMS dependent regions cannot use z/OS checkpoint restart, specify JOURNAL=NO for the JES2 initiator procedure and the JES3 class associated with the dependent regions execution class. You can also specify the JOURNAL= on the JES3 //*MAIN statement for dependent regions executing as jobs.

Application managed text units

The application can manage the dynamic descriptor text units instead of IMS. If the application manages the text units, overhead for parsing and text unit build can be reduced.

Use the `SETO` call to have IMS build dynamic descriptor text units. After they are built, these text units can be used with subsequent `CHNG` calls to define the print characteristics for a data set.

To reduce overhead by managing the text units, the text units should be used with several change calls. An example of this is a wait-for-input (WFI) transaction. The same data set attributes can be used for all print data sets. For the first message processed, the application uses the SETO call to build the text units for dynamic descriptors and a subsequent CHNG call with the TXTU= parameter referencing the prebuilt text units. For all subsequent messages, only a CHNG call using the prebuilt text units is necessary.

Note: No testing has been done to determine the amount of overhead that might be saved using prebuilt text units.

BSAM I/O area

The I/O area for spool messages can be very large. It is not uncommon for the area to be 32 KB in length. To reduce the overhead incurred with moving large buffers, IMS attempts to write to the spool data set from the application's I/O area.

BSAM does not support I/O areas in 31-bit storage for SYSOUT files. If IMS finds that the application's I/O area is in 31-bit storage:

- A work area is obtained from 24-bit storage.
- The application's I/O area is moved to the work area.
- The spool data set is written from the work area.

If the application's I/O area can easily be placed in 24-bit storage, the need to move the I/O area can be avoided and possible performance improvements achieved.

Note: No testing has been done to determine the amount of performance improvement possible.

Since a record can be written by BSAM directly from the application's I/O area, the area must be in the format expected by BSAM. The format must contain:

- Variable length records
- A Block Descriptor Word (BDW)
- A Record Descriptor Word (RDW)

IMS Spool API application coding considerations

Your application can send data to a JES Spool or Print server using a print data set. You can set options for message integrity and recovering data when failures occur.

Print data formats

The IMS Spool API attempts to provide a transparent interface for the application to insert data to the JES spool. The data can be in line, page, IPDS, AFPDS, or any format that can be handled by a JES Spool or Print server that processes the print data set. The IMS Spool API does not translate or otherwise modify the data inserted to the JES spool.

Message integrity options

The IMS Spool API provides support for message integrity.

This is necessary because IMS cannot properly control the disposition of a print data set when:

- IMS abnormal termination does not execute because of a hardware or software problem.
- A dynamic deallocation error exists for a print data set.

- Logic errors are in the IMS code.

In these conditions, IMS might not be able to stop the JES subsystem from printing partial print data sets. Also, the JES subsystems do not support a two-phase sync point.

Print disposition

The most common applications using Advanced Function Printing (AFP) are TSO users and batch jobs. If any of these applications are creating print data sets when a failure occurs, the partial print data sets will probably print and be handled in a manual fashion. Many IMS applications creating print data sets can manage partial print data sets in the same manner. For those applications that need more control over the automatic printing by JES of partial print data sets, the IMS Spool API provides the following integrity options. However, these options alone might not guarantee the proper disposition of partial print data sets. These options are the **b** variable following the IAFP keyword used with the `CHNG` call.

b=0
Indicates no data set protection

This is probably the most common option. When this option is selected, IMS does not do any special handling during allocation or deallocation of the print data set. If this option is selected, and any condition occurs that prevents IMS from properly disposing the print data set, the partial data set probably prints and must be controlled manually.

b=1
Indicates SYSOUT HOLD protection

This option ensures that a partial print data set is not released for printing without a JES operator taking direct action. When the data set is allocated, the allocation request indicates to JES that this print data set be placed in SYSOUT HOLD status. The SYSOUT HOLD status is maintained for this data set if IMS cannot deallocate the data set for any reason. Because the print data set is in HOLD status, a JES operator must identify the partial data set and issue the JES commands to delete or print this data set.

If the print data set cannot be deleted or printed:

- Message DFS0012I is issued when a print data set cannot be deallocated.
- Message DFS0014I is issued during IMS emergency restart when an in-doubt print data set is found. The message provides information to help the JES operator find the proper print data set and effect the proper print disposition.

  Some of the information includes:

  – JOBNAME
  – DSNAME
  – DDNAME
  – A recommendation on what IMS believes to be the proper disposition for the data set (for example, printing or deleting).

By using the Spool Display and Search Facility (SDSF), you can display the held data sets, identify the in-doubt print data set by DDNAME and DSNAME, and issue the proper JES command to either delete or release the print data set.

b=2
Indicates a non-selectable destination

This option prevents the automatic printing of partial print data sets. The IMS Spool API function requests a remote destination of IMSTEMP for the data set when the data set is allocated. The JES system must have a remote destination of IMSTEMP defined so that JES does not attempt to print any data sets that are sent to the destination.

If **b=2**, the name of the remote destination for the print data set must be specified in the destination name field of the call parameter list when the CHNG call is issued. When IMS deallocates the data set at sync point, and the data set prints, IMS requests that the data set be transferred to the requested final remote destination.

If the remote destination is not defined to the JES system, a dynamic allocation failure occurs. Because this remote destination is defined as non-selectable, and if IMS is unable to deallocate the print data set and control its proper disposition, the print data set remains associated with remote destination IMSTEMP when deallocated by z/OS.

When an deallocation error occurs, message DFS0012I is issued to provide details of the deallocation error and help identify the print data set that requires operator action. When partial print data sets are left on this special remote destination, the JES operator can display all the print data sets associated with this JES destination to locate the data set that requires action. The **b=2** option simplifies the operator's task of locating partial print data sets.

Message options

The third option on the IAPF keyword controls informational messages issued by the IMS Spool API support. These messages inform the JES operator of in-doubt data sets that need action.

c=0
Indicates that no DFS0012I or DFS0014I messages are issued for the print data set. You can specify **c=0** only if **b=0** is specified.

c=m
Indicates that DFS0012I and DFS0014I messages are issued if necessary. You can specify **c=m** or if **b=1** or if **b=2**, it is the default.

Option **c** does not affect issuing message DFS0013E.

IMS emergency restart

When IMS emergency restart is performed, DFS0014I messages might be issued if IMS finds that the proper disposition of a print data set is in-doubt, as a result of the restart. This message is only issued if the message option for the print data set was requested or **c=m** on the IAFP variable. When a DFS0014I message is received, a JES operator might need to find and properly dispose of the print data set. The DFS0014I message provides a recommended disposition (that is, deletion or printing).

Destination name (LTERM) usage

The standard CHNG call parameter list contains a destination name field. For traditional message calls, this field contains the LTERM or transaction code that becomes the destination of messages sent using this alternate PCB. When ISRT calls are issued against the PCB, the data is sent to the LTERM or transaction.

However, the destination name field has no meaning to the IMS Spool API function unless **b=2** is specified following the IAFP keyword.

When **b=2** is specified:

- The name must be a valid remote destination supported by the JES system that receives the print data sets.
- If the name is not a valid remote destination, an error occurs during dynamic deallocation.

If any option other than **2** is selected, the name is not used by IMS.

The LTERM name appears in error messages and log records. Use a name that identifies the routine creating the print data set. This information can aid in debugging application program errors.

Understanding parsing errors

When you are diagnosing multiple parsing error return codes, the first code returned is usually the most informative.

Keywords

The `CHNG` and `SETO` calls have two types of keywords. The type of keyword determines what type of keyword validation IMS should perform. The keyword types are:

- Keywords valid for the calls (for example, IAFP, PRTO, TXTU, and OUTN)
- Keywords valid as operands of the PRTO keyword (for example CLASS and FORMS).

Incorrectly specified length fields can cause errors when IMS checks for valid keywords. When IMS is checking the validity of keywords on the `CHNG` and `SETO` calls, one set of keywords is valid. When IMS is checking the validity of keywords on the PRTO keyword, another set of keywords is valid. For this reason, incorrectly specified length fields can cause a scan to terminate prematurely, and keywords that appear to be valid are actually invalid because of where they occur in the call list. IMS might report that a valid keyword is invalid if it detects a keyword with an incorrect length field or a keyword that occurs in the wrong place in the call list.

Status codes

The status code returned for the call can also suggest the location of the error. Although exceptions exist, generally, an AR status code is returned when the keyword is invalid for the call. An AS status code is returned when the keyword is invalid as a PRTO option.

Error codes

This topic contains information on Spool API error codes that your application program can receive. The topic “Diagnosis examples” contains examples of errors and the resulting error codes provided to the application program.

Error Code
Reason

(0002) Unrecognized option keyword.

Possible reasons for this error are:

- The keyword is misspelled.
- The keyword is spelled correctly but is followed by an invalid delimiter.
- The length specified field representing the PRTO is shorter than the actual length of the options.
- A keyword is not valid for the indicated call.

(0004) Either too few or too many characters were specified in the option variable. An option variable following a keyword in the options list for the call is not within the length limits for the option.

(0006) The length field (LL) in the option variable is too large to be contained in the options list. The options list length field (LL) indicates that the options list ends before the end of the specified option variable.

(0008) The option variable contains an invalid character or does not begin with an alphabetic character.

(000A) A required option keyword was not specified.

Possible reasons for this error are:

- One or more additional keywords are required because one or more keywords were specified in the options list.
- The specified length of the options list is more than zero but the list does not contain any options.

(000C) The specified combination of option keywords is invalid. Possible causes for this error are:

- The keyword is not allowed because of other keywords specified in the options list.
- The option keyword is specified more than once.

(000E) IMS found an error in one or more operands while it was parsing the print data set descriptors. IMS usually uses z/OS services (SJF) to validate the print descriptors (PRTO= option variable). When IMS calls SJF, it requests the same validation as for the TSO OUTDES command. Therefore, IMS is insensitive to changes in output descriptors. Valid descriptors for your system are a function of the MVS release level. For a list of valid descriptors and proper syntax, use the TSO HELP OUTDES command.

IMS must first establish that the format of the PRTO options is in a format that allows the use of SJF services. If it is not, IMS returns the status code **AS**, the error code (000E), and a descriptive error message. If the error is detected during the SJF process, the error message from SJF will include information of the form (R.C.=xxxx,REAS.=yyyyyyyy), and an error message indicating the error.

Related reading: For more information on SJF return and reason codes, see *z/OS MVS Programming: Authorized Assembler Services Guide* .

The range of some variables is controlled by the initialization parameters. Values for the maximum number of copies, allowable remote destination, classes, and form names are examples of variables influenced by the initialization parameters.

Diagnosis examples

The following examples illustrate mistakes that can generate the various spool API error codes, and diagnosis of the problems.

Some length fields are omitted when they are not necessary to illustrate the example. The feedback and options lists that are shown on multiple lines are contiguous.

Error code (0002)

Two examples of the error code 0002 are shown in this section.

For the first example the options list contains both the keywords PRTO and TXTU. The keyword, TXTU, is invalid for the SETO call.

```
CALL = SETO
  OPTIONS LIST = PRTO=04DEST(018),CLASS(A),TXTU=SET1
  FEEDBACK = TXTU(0002)
  STATUS CODE = AR
```

For the second example, the length field of the PRTO options is too short to contain all of the options. This means that IMS finds the COPIES and FORMS keywords outside the PRTO options list area and indicates that they are invalid on the CHNG call.

```
CALL = CHNG
  OPTIONS LIST = IAFP=N0M,PRTO=0FDEST(018),LINECT(200),CLASS(A),
                                  COPIES(80),FORMS(ANS)
  FEEDBACK = COPIES(0002),FORMS(0002)
  STATUS CODE = AR
```

Error code (0004)

For this example, the operand for the OUTN keyword is 9 bytes long and exceeds the maximum value for the OUTPUT JCL statement.

```
CALL = CHNG
  OPTIONS LIST = IAFP=N0M,OUTN=OUTPUTDD1
  FEEDBACK = OUTN(0004)
  STATUS CODE = AR
```

Error code (0006)

The length of the options list for this call is too short to contain all of the operands of the PRTO keyword.

This example shows an options list that is X'48' bytes long and is the correct length. The length field of the PRTO keyword incorrectly indicates a length of X'5A'. The length of the PRTO options exceeds the length of the entire options list so IMS ignores the PRTO keyword and scans the rest of the options list for valid keywords. The feedback area contains the PRTO(0006) code (indicating a length error) and the (0002) code (indicating that the PRTO keywords are in error). This is because the keywords beyond the first PRTO keyword, up to the length specified in the options list length field, have been scanned in search of valid keywords for the call. The status code of AR indicates that the keywords are considered invalid for the call and not the PRTO keyword.

```
CALL = CHNG
                0400              05
  OPTIONS LIST = 0800IAFP=N0M,PRTO=0ADEST(018),LINECT(200),CLASS(A),
                                  COPIES(3),FORMS(ANS)
  FEEDBACK = PRTO(0006),LINECT(0002),CLASS(0002),COPIES(0002),
                        FORMS(0002)
  STATUS CODE = AR
```

Error code (0008)

In this example, the message option of the IAFP keyword is incorrectly specified as "Z".

```
CALL = CHNG
                                00
  OPTIONS LIST = IAFP=N0Z,PRTO=0BDEST(018)
  FEEDBACK = IAFP(0008) INVALID VARIABLE
  STATUS CODE = AR
```

Error code (000A)

In this example, the valid keyword TXTU is specified, but the call also requires that the IAFP keyword be specified if the TXTU keyword is used.

```
CALL = CHNG
  OPTIONS LIST = TXTU=SET1
  FEEDBACK = TXTU(000A)
  STATUS CODE = AR
```

Error code (000C)

The **AR** status code is returned with the (000C) error code. This implies that the problem is with the call options and not with the PRTO options.

The call options list contains the PRTO and TXTU keywords. These options cannot be used in the same options call list.

```
CALL = CHNG
                                00
  OPTIONS LIST = IAFP=A00,PRTO=0BCOPIES(3),TXTU=SET1
  FEEDBACK = TXTU(000C)
  STATUS CODE = AR
```

Error code (000E)

In this example, the COPIES parameter has the incorrect value "RG" specified as one of its operands. The error message indicates that the values for these operands must be numeric.

```
CALL = CHNG
                                01
  OPTIONS LIST = IAFP=A00,PRTO=0BCOPIES((3),(8,RG,18,80))
  FEEDBACK = PRTO(000E) (R.C.=0004,REAS.=00000204) COPIES/RG VALUE
                        MUST BE NUMERIC CHARACTERS
  STATUS CODE = AS
```

This example includes an invalid PRTO operand. The resulting reason code of X'000000D0' indicates that the XYZ operand is invalid.

```
CALL = CHNG
                                00
  OPTIONS LIST = IAFP=A00,PRTO=0AXYZ(018)
  FEEDBACK = PRTO(000E) (R.C.=0004,REAS.=000000D0) XYZ
  STATUS CODE = AS
```

Understanding allocation errors

The IMS Spool API interface defers dynamic allocation of the print data set until data is actually inserted into the data set. Incorrect data set print options on the CHNG or SETO call can cause errors during dynamic allocation. The print data set options can be parsed during the processing of the CHNG and SETO calls but some things, for example the *destination name* parameter, can be validated only during dynamic allocation.

If one of the print options is incorrect and dynamic allocation fails when the IMS performs the first insert for the data set, IMS returns a **AX** status code to the ISRT call. IMA also issues message DFS0013E and writes a diagnostic log record (67D0) that you can use to evaluate the problem. The format of the error message indicates the type of service that was invoked and the return and reason codes that were responsible for the error. The error message can indicate these services:

DYN MVS dynamic allocation (SVC99)

OPN MVS data set open

OUT MVS dynamic output descriptors build (SVC109)

UNA MVS dynamic unallocation (SVC99)

WRT MVS BSAM write

If the DFS0013E message indicates an error return code from any of these services, you should consult the corresponding MVS documentation for more information on the error code. If the service is for dynamic allocation, dynamic unallocation, or dynamic output descriptor build, see *z/OS MVS Programming: Authorized Assembler Services Guide* for the appropriate return and reason codes.

One common mistake is the use of an invalid destination or selection of integrity option 2 (non-selectable destination) when the destination of IMSTEMP has not been defined to JES. If you specify an invalid destination in the **destination name** parameter, the call will result in a dynamic unallocation error when IMS unallocates the print data set.

Understanding dynamic output for print data sets

IMS can use the z/OS services for Dynamic Output (SVC109) for print data sets. IMS uses this service to specify the attributes provided by the application for the print data sets being created. The service can be used on the CHNG call with the PRTO, TXTU, and OUTN options.

Related reading: For more information, see *z/OS MVS Programming: Assembler Services Guide*.

CHNG call with PRTO option

When you use the CHNG call and PRTO option, IMS activates SJF to verify the print options to call z/OS services for Dynamic Output. This creates the output descriptors that are used when the print data set is allocated. This is the simplest way for the application to provide print data set characteristics. However, it also uses the most overhead because parsing must occur for each CHNG call. If your application is WFI or creates multiple data sets with the same print options, use another option to reduce the parsing impact. You must specify the IAFP option keyword with this option.

CHNG call with TXTU option

If your application can manage the text units necessary for Dynamic Output, then you can avoid parsing for many of the print data sets. You can do this in one of two ways:

- The application can build the text unit in the necessary format within the application area and pass these text units to IMS with the CHNG call and TXTU option.
- The application can provide the print options to IMS with a SETO call and provide a work area for the construction of the text units. After z/OS has finished parsing and text construction, the work area passed will contain the text units necessary for Dynamic Output after a successful SETO call. The application must not relocate this work area because the work area contains address sensitive information.

Regardless of the method the application uses to manage the text units, applications that can reuse the text units can often achieve better performance by using the TXTU option on the CHNG call.

You must specify the IAFP option keyword with this option.

CHNG call with OUTN option

The dependent region JCL can contain OUTPUT JCL statements. If your application can use this method, you can use the CHNG call and OUTN option to reference OUTPUT JCL statements. When you use the OUTN option, IMS will reference the OUTPUT JCL statements at dynamic allocation. JES will obtain the print data set characteristics from the OUTPUT JCL statement. You must specify the IAFP option keyword with this option.

Sample programs using the Spool API

The Spool API provides functions that allow an application program to write data to the IMS Spool using the same techniques for sending data to native IMS printers.

The Spool API provides functions such as error checking for invalid OUTDES parameters. Error checking makes application programs more complex. To simplify these application programs, develop a common routine to manage error information, then make the diagnostic information from the Spool API available for problem determination.

The sample programs in this section shows how DL/I calls can be coded to send data to the IMS Spool. Only the parts of the application program necessary to understand the DL/I call formats are included. The examples are in assembler language.

Application PCB structure

The application PCBs are as follows:

I/O PCB

ALTPCB1

ALTPCB2

ALTPCB3

ALTPCB4

GU call to I/O PCB

IMS application programs begin with initialization and a call to the I/O PCB to obtain the input message. The following code example shows how to issue a GU call to the I/O PCB.

After completing the GU call to the I/O PCB, the application program prepares output data for the IMS Spool.

Issuing a GU call to the I/O PCB

```
*********************************************************************
*       ISSUE GU ON IOPCB                                           *
*********************************************************************
         L     9,IOPCB             I/O PCB ADDRESS
         LA    9,0(9)
         MVC   FUNC,=CL4'GU'       GU FUNCTION
         CALL  ASMTDLI,(FUNC,(9),IOA1),VL
         BAL   10,STATUS           CHECK STATUS
*        ADDITIONAL PROGRAM LOGIC HERE
FUNC     DC    CL4' '
IOA1     DC    AL2(IOA1LEN),AL2(0)
TRAN     DS    CL8                 TRANSACTION CODE AREA
DATA     DS    CL5                 DATA STARTS HERE
         DC    20F'0'
IOA1LEN  EQU   *-IOA1
```

CHNG call to alternate PCB

In the same way that other programs specify the destination of the output using the CHNG call, this program specifies the IMS Spool as the output destination. For a native IMS printer, the DEST NAME parameter identifies the output LTERM name. When a CHNG call is issued that contains the IAFP= keyword, the DEST NAME parameter is used only if integrity option '2' is specified. If option '2' is not specified, the DEST NAME parameter can be used by the application program to identify something else, such as the routine producing the change call. The destination for the print data set is established using a combination of initialization parameters or OUTDES parameters.

The following code example shows how to issue a CHNG call to the alternate modifiable PCB.

After the CHNG call is issued, the application program creates the print data set by issuing ISRT calls.

Issuing a CHNG call to the alternate modifiable PCB

```
*********************************************************************
*       ISSUE CHNG ON ALTPCB4                                       *
*********************************************************************
         L     9,ALTPCB4           ALT MODIFIABLE PCB
         LA    9,0(9)              CLEAR HIGH BYTE/BIT
         MVC   FUNC,=CL4'CHNG'     CHNG FUNCTION
         CALL  ASMTDLI,(FUNC,(9),DEST2,OPT1,FBA1),VL
         BAL   10,STATUS           CHECK STATUS OF CALL
*        ADDITIONAL PROGRAM LOGIC HERE
FUNC     DC    CL4' '
DEST2    DC    CL8'IAFP1'          LTERM NAME
*
         DC    C'OPT1'             OPTIONS LIST AREA
```

```
OPT1     DC    AL2(OPT1LEN),AL2(0)
         DC    C'IAFP='
OCC      DC    C'M'                DEFAULT TO MACHINE CHAR
OOPT     DC    C'1'                DEFAULT TO HOLD
OMSG     DC    C'M'                DEFAULT TO ISSUE MSG
         DC    C','
         DC    C'PRTO='
PRTO1    EQU   *
         DC    AL2(PRTO1LEN)
         DC    C'COPIES(2),CLASS(T),DEST(RMT003)'
PRTO1LEN EQU   *-PRTO1
         DC    C' '
OPT1LEN  EQU   *-OPT1
*
FBA1     DC    AL2(FBA1LEN),AL2(0)
         DC    CL40' '
FBA1LEN  EQU   *-FBA1
```

ISRT call to alternate PCB

Once the IMS Spool is specified as the destination of the PCB, ISRT calls can be issued against the alternate PCB.

The following code example shows how to issue the ISRT call to the alternate modifiable PCB.

The print data streams can be stored in databases or generated by the application, depending on the requirements of the application program and the type of data set being created.

Issuing an ISRT call to the alternate modifiable PCB

```
*********************************************************************
*       ISSUE ISRT TO ALTPCB4                                       *
*********************************************************************
         L     9,ALTPCB4           ALT MODIFIABLE PCB
         LA    9,0(9)              CLEAR HIGH BYTE/BIT
         MVC   FUNC,=CL4'ISRT'     ISRT FUNCTION
         CALL  ASMTDLI,(FUNC,(9),IOA2),VL
         BAL   10,STATUS           CHECK STATUS OF CALL
*        ADDITIONAL PROGRAM LOGIC HERE
FUNC     DC    CL4' '
IOA2     DC    AL2(IOA2LEN),AL2(0)
IOA21    DC    AL2(MSG2LEN),AL2(0)
         DC    C' '                CONTROL CHARACTER
         DC    C'MESSAGE TO SEND TO IMS SPOOL'
MSG2LEN  EQU   *-IOA21
IOA2LEN  EQU   *-IOA2
```

Program termination

After the calls are issued, the program sends a message back to originating terminal, issues a GU call to the I/O PCB, or terminates normally.

Chapter 26. IMS Message Format Service

The IMS Message Format Service (MFS) is a facility of the IMS Transaction Manager environment that formats messages to and from terminal devices, so that IMS application programs do not deal with device-specific characteristics in input or output messages.

In addition, MFS formats messages to and from user-written programs in remote controllers and subsystems, so that application programs do not deal with transmission-specific characteristics of the remote controller.

MFS uses control blocks you specify to indicate to IMS how input and output messages are arranged.

- For input messages, MFS control blocks define how the message sent by the device to the application program is arranged in the program's I/O area.
- For output messages, MFS control blocks define how the message sent by the application program to the device is arranged on the screen or at the printer. Data that appears on the screen but not in the program's I/O area, such as a literal, can also be defined.

In IMS Transaction Manager systems, data passing between the application program and terminals or remote programs can be edited by MFS or basic edit. Whether an application program uses MFS depends on the type of terminals or secondary logical units (SLUs) your network uses.

Restriction: MFS does not support message formatting for LU 6.2 devices.

Advantages of using MFS

By using MFS, you can simplify the developing and maintaining of terminal-oriented applications, and improve online performance by using control blocks for online processing.

Simplify development and maintenance

To simplify IMS application development and maintenance, MFS performs many common application program functions and gives application programs a high degree of independence from specific devices or remote programs.

With the device independence offered by MFS, one application program can process data to and from multiple device types while still using their different capabilities. Thus, MFS can minimize the number of required changes in application programs when new terminal types are added.

MFS makes it possible for an application program to communicate with different types of terminals without having to change the way it reads and builds messages. When the application receives a message from a terminal, how the message appears in the program's I/O area is independent of what kind of terminal sent it; it depends on the MFS options specified for the program. If the next message the application receives is from a different type of terminal, you do not need to do anything to the application. MFS shields the application from the physical device

© Copyright IBM Corp. 1974, 2015

that is sending the message in the same way that a DB program communication block (PCB) shields a program from what the data in the database actually looks like and how it is stored.

Other common functions performed by MFS include left or right justification of data, padding, exits for validity checking, time and date stamping, page and message numbering, and data sequencing and segmenting. When MFS assumes these functions, the application program handles only the actual processing of the message data.

The following figure shows how MFS can make an application program device-independent by formatting input data from the device or remote program for presentation to IMS, and formatting the application program data for presentation to the output device or remote program.

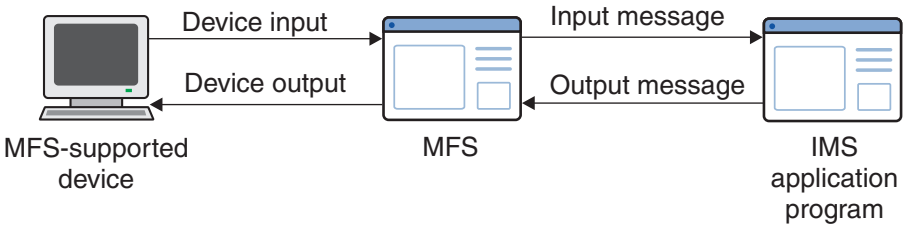


Figure 82. Message formatting using MFS

Improve online performance of a terminal

MFS also improves online performance of a terminal-oriented IMS by using control blocks designed for online processing. The MFS control blocks are compiled offline, when the IMS Transaction Manager system is not being executed, from source language definitions. MFS can check their validity and make many decisions offline to reduce online processing. In addition, during online processing, MFS uses look-aside buffering of the MFS control blocks to reduce CPU and channel costs of input/output activity.

Because MFS control blocks are reentrant and can be used for multiple applications, online storage requirements are reduced. Optional real storage indexing and anticipatory fetching of the control blocks can also reduce response time. Further performance improvements can be gained when IMS is generated for z/OS, since multiple I/O operations can execute concurrently to load the format blocks from the MFS format library.

In addition, MFS uses z/OS paging services; this helps to reduce page faults by the IMS control region task.

MFS can reduce use of communication lines by compressing data and transmitting only required data. This reduces line load and improves both response time and device performance.

MFS control blocks

There are four types of MFS control blocks that you specify to format input and output for the application program and the terminal or remote program.

The four types are:

Message Output Descriptors (MODs)
Define the layout of messages MFS receives from the application program.

Device Output Formats (DOFs)
Describe how MFS formats messages for each of the devices the program communicates with.

Device Input Formats (DIFs)
Describe the formats of messages MFS receives from each of the devices the program communicates with.

Message Input Descriptors (MIDs)
Describe how MFS further formats messages so that the application program can process them.

Throughout this information, the term "message descriptors" refers to both MIDs and MODs. The term "device formats" refers to both DIFs and DOFs.

Each MOD, DOF, DIF and MID deals with a specific message. There must be a MOD and DOF for each unique message a program sends, and a DIF and MID for each unique message a program receives.

MFS examples

One way to understand the relationship between the MFS control blocks is to look at a message from the time a user enters it at the terminal to the time the application program processes the message and sends a reply back to the terminal. Though MFS can be used with both display terminals and printer devices, for clarity in this example, a display terminal is being used.

The following figure shows the relationships between the MFS control blocks.

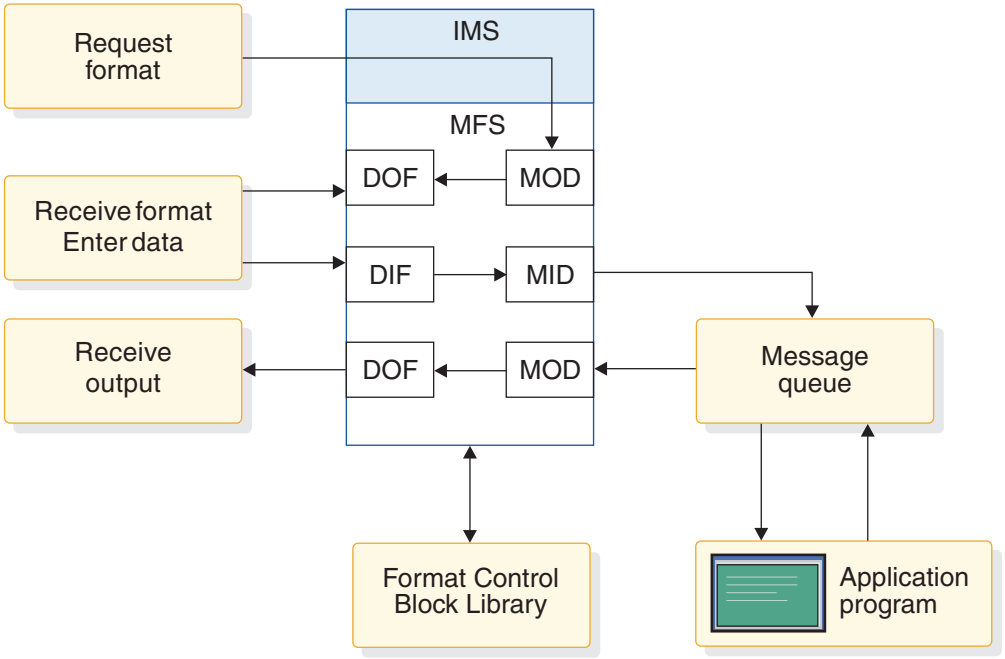


Figure 83. MFS control block relationships

Looking at payroll records

Suppose your installation has a message processing program used to view employee payroll records. From a display terminal, issue the IMS format command (/FORMAT), and the MOD name. This formats the screen in the way defined by the MOD written by the MFS programmer. When you enter the MOD name, the screen contains only literals and no output data from the application program. At this stage, no application program is involved. (For more information about /FORMAT, see *IMS Version 11 Commands, Volume 1: IMS Commands A-M*.)

In this example, suppose the name of the MOD that formats the screen for this application is PAYDAY. Enter this command:

```
/FORMAT PAYDAY
```

IMS locates the MFS MOD control block with the name PAYDAY and arranges the screen in the format defined by the DOF.

The following figure shows how this screen looks.

```
                        *EMPLOYEE PAYROLL*
                        ******************


FIRST NAME:                              LAST NAME:
EMPLOYEE NO:



INPUT:
```

Figure 84. PAYDAY screen, formatted by DOF

The DOF defines a terminal format that asks you to give the employee's name and employee number. PAYUP is the transaction code associated with the application that processes this information. When you enter the MOD name, the transaction code is included in the first screen format displayed. This means that you do not need to know the name of the program that processes the data; you only need the name of the MOD that formats the screen.

After the screen format is displayed, you can enter the information. There are four stages to sending a message to the program and receiving the reply:

1. Enter the information at the terminal. For this example, enter the prompted information.

   The following figure shows how this screen looks after information is entered.

```
                        *EMPLOYEE PAYROLL*
                        ******************


FIRST NAME: Joe                          LAST NAME: Blutzen
EMPLOYEE NO: 60249



INPUT:
```

Figure 85. PAYDAY screen, with filled input fields

2. When IMS receives this data, MFS uses the DIF and the MID control blocks to translate the data from the way it was entered on the terminal screen to the way that the application program is expecting to receive it. The DIF control block tells MFS the format of the data to come in from the terminal. The MID control block tells MFS how the application program expects to receive the data. When the application program issues a message call, IMS places the "translated" message in the program's I/O area.

   When the application receives the message in its I/O area, the message looks like this:

   ```
   PAYUP JOE BLUTZEN 60249
   ```

   "PAYUP" is the transaction code. The name of the logical terminal does not appear in the message itself; IMS places it in the first field of the I/O PCB.

3. The application program processes the message, including any required database access, and builds the output message in the application program's I/O area. After retrieving the information from the database, the program builds the output message segment for the employee, with social security and rate of pay information. The application program's I/O area contains:

   ```
   LLZZJOE BLUTZEN 60249532596381150.00
   ```

   The LL is a 2-byte field in MFS messages that indicates the length of the field. How the LL field is defined depends on what programming language used to write the application program. For the AIBTDLI, ASMTDLI, CEETDLI, or PASTDLI interfaces, the LL field must be defined as a binary half word. For the PLITDLI interface, the LL field must be defined as a binary fullword. The value provided in the PLITDLI interface must represent the actual segment length minus 2 bytes.

   The ZZ is a 2-byte length field in MFS messages that contains the MFS formatting option that is being used to format the messages to and from the application program. MFS options are discussed in further detail in the topic "Input Message Formatting Options" in *IMS Version 11 Application Programming APIs*.

4. When the application program sends the message back to the terminal, MFS translates the message again, this time from the application program format to the format in which the terminal expects the data.

   The MOD tells MFS the format that the message will be in when it comes from the application program's I/O area. The DOF tells MFS how the message is supposed to look on the terminal screen. MFS translates the message and IMS displays the translated message on the terminal screen.

   The following figure shows how the screen looks.

```
                        *EMPLOYEE PAYROLL*
                        ******************


FIRST NAME: Joe                       LAST NAME: Blutzen
EMPLOYEE NO: 60249
SOC SEC NO: 532-59-6381
RATE OF PAY: $150.00


INPUT:
```

Figure 86. PAYDAY screen, output formatted by DOF and displayed

Listing a subset of employees

Suppose you have an MPP that answers this request:

> List the employees who have the skill "ENGINEER" with a skill level of "3." List only those employees who have been with the firm for at least 4 years.

To enter the request from a display terminal, issue the format command (/FORMAT) and the MOD name. This formats the screen in the way defined by the MOD you supply. When you enter the MOD name, the screen contains only literals and no output data from an application program. At this stage, an MPP is not involved. Suppose the name of the MOD that formats the screen for this request is LE, for "locate employee." Enter this:

```
/FORMAT LE
```

IMS locates the MFS MOD control block with the name LE and arranges your screen in the format defined by the DOF. Your screen then looks like this:

```
SKILL
LEVEL
YEARS
      LOCEMP
```

The DOF defines a terminal format that asks you to qualify your request for an employee by giving the skill, level, and number of years of service of the employee you want. LOCEMP is the transaction code that is associated with the MPP that can process this request. When you enter the MOD name, the transaction code is included in the first screen format that is displayed for you. This means that you do not need the name of the program that processes your request; you only need the name of the MOD that formats the screen.

After the screen format is displayed, you can enter your request. There are four stages in sending a message to the program and receiving the reply.

1. Enter the information at the terminal. In this example, enter the values of the qualifications that IMS has given you on the screen: the skill is "eng" (engineer), the skill level is "3," and the number of years with the firm is "4".

   After you enter your request, your screen contains this data:

   ```
   SKILL ENG
   LEVEL 3
   YEARS 4
         LOCEMP
   ```

2. When IMS receives this data, MFS uses the DIF and the MID control blocks to translate the data from the way you entered it on the terminal screen to the way that the application program is expecting to receive it. The DIF control block tells MFS how the data is going to come in from the terminal. The MID control block tells MFS how the application program is expecting to receive the data. When the application program issues a GU call to the I/O PCB, IMS places the "translated" message in the program's I/O area.

   When the MPP receives the message in its I/O area, the message looks like this:

   ```
   LOCEMP  ENG0304
   ```

   "LOCEMP" is the transaction code. The name of the logical terminal does not appear in the message itself; IMS places it in the first field of the I/O PCB.

3. The MPP processes the message, including any required database access, and builds the output message in the MPP's I/O area.

Suppose more than one employee meets these qualifications. The MPP can use one message segment for each employee. After retrieving the information from the database, the program builds the output message segment for the first employee. The program's I/O area contains:

```
LLZZJONES,CE   3294
```

When the program sends the second segment, the I/O area contains:

```
LLZZBAKER,KT   4105
```

4. When the application program sends the message back to the terminal, MFS translates the message again, this time from the application program format to the format in which the terminal expects the data.

   The MOD tells MFS the format that the message will be in when it comes from the application program's I/O area. The DOF tells MFS how the message is supposed to look on the terminal screen. MFS translates the message and IMS displays the translated message on the terminal screen. The screen then contains the following data:

```
SKILL    ENG
NAME     NO
JONES,CE 3294
BAKER,KT 4105
```

Related concepts:

"Relationship between MFS control blocks and screen format"

Relationship between MFS control blocks and screen format

Use the control blocks in the MFS source language to define the formats that you see at the device.

The standard way for an end-user or operator to receive an initial format is to request it with a /FORMAT command, specifying the name of a MOD. In the following code example, the label on the MOD is PAYDAY. This MOD contains the parameter SOR=PAYF, which points to a device output format, or DOF, with the same label.

The initial DOF also becomes the format for device input. Therefore, if you specify DIV TYPE=INOUT in the DOF, a device input format (DIF) is also generated. In the sample code, PAYF is both a DOF and a DIF, since it also describes the format of the next input. The final output message can be displayed with a format that is specified for output only and no DIF is generated.

Both the MOD and the MID point to the same DOF, thus establishing the relationship between device-related and message-related control blocks.

For output, MFS moves fields defined in a MOD to fields on the screen defined by a DOF. When a field definition is coded (MFLD) in a MOD, it is given a label. The same label is used in the coding of the device field (DFLD) in the DOF, defining where the field appears on the screen.

MFS moves data fields from output messages to screen fields; this is referred to as *mapping*. For input, MFS moves modified screen fields to data fields in the input message for the program by mapping identically labeled fields in the DIF and MID.

For more detailed information on specifying these control blocks, see *IMS Version 11 Database Utilities*.

The MFS control blocks are generated from the source statements like those in the following code example during execution of the MFS Language utility. The control blocks are stored in the various MFS libraries.

The sample code is designed for a 3270 display.

Sample MFS control block coding

DOF/DIF

```
PAYF     FMT
         DEV       TYPE=(3270,2),FEAT=IGNORE,DSCA=X'00A0'
         DIV       TYPE=INOUT
         DPAGE     CURSOR=((5,15))
         DFLD      '**********************',POS=(1,21)
         DFLD      '*  EMPLOYEE PAYROLL  *',POS=(2,21)
         DFLD      '**********************',POS=(3,21)
         DFLD      'FIRST NAME:',POS=(5,2)
FNAME    DFLD      POS=(5,15),LTH=16
         DFLD      'LAST NAME:',POS=(5,36)
LNAME    DFLD      POS=(5,48),LTH=16
         DFLD      'EMPLOYEE NO:',POS=(7,2)
EMPNO    DFLD      POS=(7,16),LTH=6
         DFLD      'SOC SEC NO:',POS=(9,2)
SSN      DFLD      POS=(9,15),LTH=11
         DFLD      'RATE OF PAY: $',POS=(11,2)
RATE     DFLD      POS=(11,17),LTH=9
         DFLD      'INPUT:',POS=(16,2)
INPUT    DFLD      POS=(16,10),LTH=30
         FMTEND
```

MID

```
PAYIN    MSG       TYPE:INPUT,SOR=(PAYF,IGNORE)
         SEG
         MFLD      'PAYUP '     SUPPLIES TRANCODE
         MFLD      LNAME,LTH=16
         MFLD      FNAME,LTH=16
         MFLD      EMPNO,LTH=6
         MFLD      SSN,LTH=11
         MFLD      RATE,LTH=9
         MFLD      INPUT,LTH=30,JUST=R,FILL=C'0'
         MSGEND
```

MOD

```
PAYDAY   MSG       TYPE:OUTPUT,SOR=(PAYF,IGNORE)
         SEG
         MFLD      LNAME,LTH=16
         MFLD      FNAME,LTH=16
         MFLD      EMPNO,LTH=6
         MFLD      SSN,LTH=11
         MFLD      RATE,LTH=9
         MFLD      INPUT,LTH=30,JUST=R,FILL=C'0'
         MSGEND
```

Related reference:

"MFS examples" on page 463

Overview of MFS components

IMS Message Format Service (MFS) components include three utilities, a message editor, and two pool managers.

MFS utilities

You can use the MFS utilities for multiple service and generation purposes:

- MFS Device Characteristics Table utility (DFSUTB00): Define new screen sizes in a descriptor member of the IMS.PROCLIB library without completing an IMS system definition.
- MFS Language utility (DFSUPAA0): Create and store the MFS control blocks.
- MFS Service utility (DFSUTSA0): Control and maintain MFS intermediate text blocks and control blocks after they are processed and stored by the MFS Language utility (DFSUPAA0).

In addition to the using the MFS utilities to update MFS libraries, you can also use the IMS online change function. You can modify control block libraries while the IMS control region is executing.

MFS message editor

Use the MFS message editor to formats messages according to the control block specifications generated by the MFS Language utility from control statement definitions that you enter.

MFS pool managers

You can customize the functions of the following MFS pool managers:

- MFS pool manager: MFS tries to minimize I/O to the format library by keeping referenced blocks in storage. This storage is managed by the MFS pool manager. You can use the INDEX function of the MFS Service utility to customize this function by constructing a list of the directory addresses for specified format blocks. This list eliminates the need for IMS to read the data set directory before it fetches a block.
- MFSTEST pool manager: If you use the MFSTEST facility, MFS control blocks are managed by the MFSTEST pool manager. The communication line buffer pool space allowed for MFS testing is specified during system definition, but the space can be changed when the IMS control region is initialized. This space value is the maximum amount used for MFSTEST blocks at any one time. The space value is not a reserved portion of the pool.

Related concepts:

Making online changes (System Administration)

MFS components (Communications and Connections)

Use of the message format buffer pool (System Definition)

Related reference:

MFS Language utility (DFSUPAA0) (System Utilities)

MFS Service utility (DFSUTSA0) (System Utilities)

MFS Device Characteristics Table utility (DFSUTB00) (System Utilities)

Devices and logical units that operate with MFS

In addition to 3270 devices, MFS operates with the 3600 and 4700 Finance Communication System (FIN), the 3770 Data Communication System, the 3790 Communication System, and with Secondary Logical Unit (SLU) types 1, 2, 6, and P. Network Terminal Option (NTO) devices are supported as secondary logical unit type 1 consoles.

The following table shows which devices or logical units can be defined for MFS operation in the IMS system by their number (3270, for example), and which can be defined by the type of logical unit to which they are assigned (SLU 1, for example).

Although the 3600 devices are included in the FIN series, you can specify them with their 36xx designations; MFS messages use the FIxx designations regardless of which form of designation you specify. In general, however, application designers and programmers using this information need to know only how the devices they are defining control blocks for have been defined to the IMS system in their installation.

Table 85. Terminal devices that operate with MFS.

Device	Devices defined by number[1]	NTO devices[2]	SLU 1	SLU 2	SLU P	LU 6.1
3180	X [3]			X [3]		
3270	X [3]			X [3]		
3290	X [3]			X [3]		
5550	X [3]			TYPE: 3270-An 3270-Ann		
3270 printers; 5553, 5557	X [3]		COMPT*n*= MFS-SCS1			
3730					X	
3767			COMPT*n*= MFS-SCS1			
3770 console, printers, print data set			COMPT*n*= MFS-SCS1		X	
3770 readers, punches, transmit data set			COMPT*n*= MFS-SCS2		X	
3790 print data set (bulk)			COMPT*n*= MFS-SCS1		COMPT*n*= MFS-SCS1 DPM-An	
3790 transmit data set			COMPT*n*= MFS-SCS2			
3790 attached 3270				X [3]		
6670						
8100					X	
8100 attached 3270			X	X [3]		
8100 attached Series/1					X	
8100 attached S/32			X			

Table 85. Terminal devices that operate with MFS (continued).

Device	Devices defined by number[1]	NTO devices[2]	SLU 1	SLU 2	SLU P	LU 6.1
8100 attached S/34					X	
8100 attached S/38			X			
Finance	X				COMPT*n*= MFS-SCS1 DPM-An	
TTY		X				
3101		X				
Other systems (IMS to IMS or IMS to other)						COMPT*n*= DPM=Bn

Notes:

1. With options= (...,MFS,...) in the TERMINAL or TYPE macro.
2. Defined with UNITYPE= on the TYPE macro and PU= on the TERMINAL macro.
3. Defaults to operate with MFS.

Logical units are defined by logical unit type or logical unit type with COMPT*n*= or TYPE= in the TERMINAL macro or ETO logon descriptor. The LU 6.1 definition refers to ISC subsystems.

The definition for SLU 1 can specify an MFS operation with SNA character strings (SCS) 1 or 2. SCS1 designates that messages are sent to a printer or the print data set or received from a keyboard in the 3770 Programmable or 3790 controller disk storage; SCS2 designates that messages are sent to or received from card I/O or a transmit data set.

Terminals defined as SLU 2 have characteristics like the 3270, and like the 3270, can be defined to operate with MFS. In general, a 3290 terminal operates like a 3270 terminal, and references to 3270 terminals in this information are applicable to 3290 devices. However, 3290 partitioning and scrolling support is provided only for 3290 devices defined to IMS as SLU 2.

Generally, the 3180 and 5550 terminals operate like a 3270 terminal, and references to 3270 terminals also apply to these devices. Likewise, the 5553 and 5557 printer devices operate like a 3270P.

Restriction: 5550 Kanji support is provided only for the 5550 terminal defined as an SLU 2 and for the 5553 and 5557 defined as SCS1 printers.

If IMS is to communicate with the user-written remote program in a 3790 or an FIN controller, the device must be defined as an SLU P. Definitions for SLU P must specify MFS operation as either MFS-SCS1 or DPM-An, where DPM means distributed presentation management and An is a user-assigned number (A1 through A15).

Most of the MFS formatting functions currently available to other devices, except specific device formatting, are available to the user-written program. Under user control, these formatting functions (such as paging) can be divided between MFS and the remote program.

Using distributed presentation management (DPM)

With distributed presentation management (DPM), formatting functions usually performed by MFS are distributed between MFS and a user-written program for SLU P devices or ISC nodes. If the 3790 or FIN controller has previously been defined to IMS by unit number, some changes must be made to convert to DPM.

With DPM, the physical terminal characteristics of the secondary logical unit do not have to be defined to MFS. MFS has to format only the messages for transmission to the user program in the remote controller or ISC node, which must assume responsibility for completing the device formatting, if necessary, and present the data to the physical device it selects.

For remote programs using DPM, the data stream passing between MFS and the remote programs can be device independent. The messages from the IMS application program can include some device control characters. If so, the IMS application program and the data stream to the remote program might lose their device independence.

If IMS is to communicate with other subsystems (such as IMS, CICS or user-written), the other subsystem must be defined as an ISC subsystem. Definitions for ISC must:

- Specify MFS operation as DPM-Bn, where Bn is a user-assigned number (B1 through B15).
- Define TYPE:LUTYPE6 on the TERMINAL macro during system definition.

DPM with ISC provides:

- Output paging on demand that allows paging to be distributed between IMS and another system
- Automatically paged output that allows MFS pages to be transmitted to another system without intervening paging requests
- Transaction routing that allows application programs to view the routing information when it is provided in the input message

Chapter 27. Callout requests for services or data

IMS applications can issue callout requests for services or data, and optionally receive responses back in the same or a different transaction, through IMS Connect and OTMA. The request for services or data is a *callout request*.

If the IMS application, after issuing the request, waits for a response in the dependent region, the request is a *synchronous callout request*. If the IMS application terminates after the request is issued and does not wait for a response in the dependent region, the request is an *asynchronous callout request*.

Each type of request is processed as follows:

Synchronous callout request
An IMS application program that runs in an IMS dependent region issues a DL/I ICAL call and waits in the dependent region to process the response. When the DL/I ICAL call is issued, IMS generates a correlation token for synchronous callout requests. This correlation token is included with the callout request and must be returned to IMS with the response to route the response back to the requesting IMS application program.

Asynchronous callout request
An IMS application program that runs in an IMS dependent region inserts the callout request to an ALTPCB queue (the ISRT ALTPCB call) and then terminates to free the dependent region. IMS does not generate a correlation token for asynchronous callout requests. If a response to the callout request is required, the correlation of the response to the callout request must be managed by the IMS application program. When IMS receives a response to an asynchronous callout request, IMS processes the response as a new transaction.

The following table summarizes the differences between synchronous and asynchronous callout requests.

Table 86. Comparison of synchronous and asynchronous callout requests

Callout process	Synchronous callout request	Asynchronous callout request
Placing the request in the OTMA hold queue	The requesting IMS application issues an ICAL call.	The requesting IMS application issues an ISRT ALTPCB call.
Status of the IMS application after the request is issued	The application waits in the dependent region for the response. Dependent regions are blocked.	The application terminates.
Message processing handling	The message processing is handled by IMS OTMA.	The message processing is handled by the IMS message queue.
Response handling	The response is correlated back to the requesting IMS application, based on the correlation token, during the same unit of work.	If there is a response, the requesting or a different IMS application must be coded to handle the response that is returned in a different transaction. The unit of work for the transaction has to commit for the asynchronous output to flow.

Related reference:

ICAL call (Application Programming APIs)

© Copyright IBM Corp. 1974, 2015

Callout request approaches

You can issue a callout request to IMS Enterprise Suite SOAP Gateway, IMS TM Resource Adapter, or to your own user-supplied IMS Connect client applications. Callout requests are routed through IMS Connect.

Using SOAP Gateway

Use SOAP Gateway to issue callout requests from IMS applications to any generic web service.

SOAP Gateway enables IMS applications as either web service providers or consumers. SOAP Gateway supports both asynchronous and synchronous callout approaches for IMS applications as web service consumers. Tooling support for SOAP Gateway is available in Rational® Developer for System z® for generating the required web service artifacts based on connection and interaction information for communicating with IMS Connect, and the language structure of the IMS applications. SOAP Gateway also provides a deployment utility to support the deployment of IMS applications as either providers or consumers of web services.

For more information about enabling IMS application callout through SOAP Gateway, see Enabling an IMS application as a web service consumer.

Using IMS TM Resource Adapter

Use IMS TM Resource Adapter Version 10 or later to issue synchronous or asynchronous callout requests from IMS applications to any message-driven bean (MDB), Enterprise JavaBeans (EJB) component, Java EE (previously known as J2EE) application, or web service.

IMS TM Resource Adapter enables Java EE applications to access IMS transactions over the Internet, as well as to issue callout requests to external Java EE applications from IMS applications that run in IMS dependent regions. The IMS TM Resource Adapter includes a runtime component for WebSphere Application Server. Tooling support for the IMS TM Resource Adapter is available in IBM Rational Application Developer for WebSphere Software, as well as various Rational and WebSphere integrated development environments (IDEs) that include the J2EE Connector (J2C) wizard.

For more information about callout support in IMS TM Resource Adapter, see Callout programming models.

Using a user-written IMS Connect TCP/IP application

You can write your own IMS Connect TCP/IP applications or use a vendor-supplied solution that uses TCP/IP and the IMS Connect protocol to retrieve callout requests. Your custom IMS Connect client application must issue a RESUME TPIPE call to an OTMA routing destination, also known as a transaction pipe (tpipe), that is defined in an OTMA destination descriptor. This tpipe holds the callout requests. Your custom IMS Connect TCP/IP application must poll the tpipe to retrieve the callout requests.

Related concepts:

Callout requests from IMS application programs (Communications and Connections)

OTMA destination descriptors (Communications and Connections)

Related reference:

DFSYDTx member of the IMS PROCLIB data set (System Definition)

ICAL call (Application Programming APIs)

Resume tpipe protocol

The resume tpipe protocol retrieves asynchronous and synchronous callout messages from IMS.

An IMS Connect client signals how long to wait for output from IMS by specifying an IRM timeout value with the IRM_TIMER field. The IRM timeout value affects the RESUME TPIPE call that is sent to IMS Connect and the ACK or NAK response message that is sent to IMS Connect.

When you use IMS TM Resource Adapter or IMS Enterprise Suite SOAP Gateway to handle the callout request from your IMS application, the communication with IMS Connect is handled for you.

Both the IMS TM Resource Adapter and SOAP Gateway listen for synchronous callout requests by continuously issuing the RESUME TPIPE call to IMS Connect. If a callout request message is on the tpipe queue, OTMA sends the callout request to IMS Connect, IMS Connect processes the message, converting the message to XML if necessary (for SOAP Gateway if using the IMS Connect XML adapter function), and then sends the message to the IMS TM Resource Adapter or SOAP Gateway.

If you have a custom IMS Connect client, you must code the client to issue a RESUME TPIPE call to retrieve the callout messages

Resume tpipe security

You can protect callout messages from unauthorized use of the RESUME TPIPE call by using either the Resource Access Control Facility (RACF), the OTMA Resume TPIPE Security exit routine (DFSYRTUX), or both.

When security is enabled, the user ID that issues the RESUME TPIPE call must be authorized to access the tpipe name that is contained in the RESUME TPIPE call message before any messages are sent to an OTMA client.

The security checking performed by RACF and the security checking performed by the DFSYRTUX exit routine are optional. If both RACF and the DFSYRTUXOTMARTUX are used, RACF is called first before giving control to the DFSYRTUX exit routine, in which case, the DFSYRTUX exit routine can override RACF, depending on your needs.

Implementing the synchronous callout function

To issue a synchronous callout request from your IMS application, issue the ICAL call and specify the OTMA descriptor name.

The ICAL call can also be issued through a REXXTDLI call, or from a Java application that runs in a JMP or JBP region. Optionally, you can also specify a timeout value (the maximum time to wait for the response to return).

Input and output messages from IMS can be 32 KB or larger per segment for a synchronous callout request.

The following diagram shows the message flow of the synchronous callout function. The request starts with an IMS application that issues an ICAL call. The response is returned to the requesting IMS application.

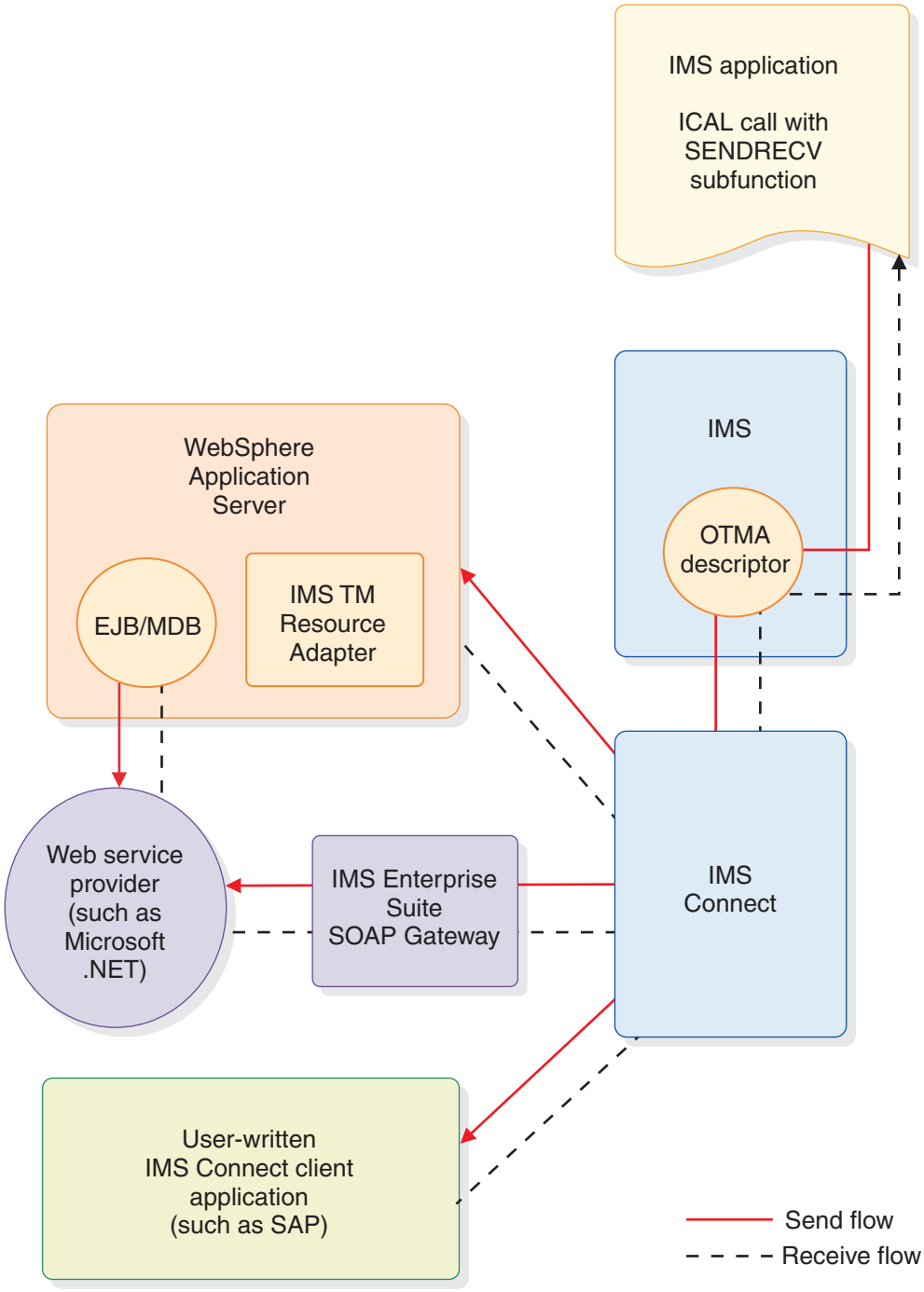


Figure 87. Message flow of the synchronous callout function

You can make concurrent ICAL calls from your IMS applications that are routed to the same or different IMS Connect destinations. By using the RESUME TPIPE call and the send-only protocol, the synchronous callout function allows the requests to

be sent and the responses to be received in different connections and threads. You can have one connection for retrieving the callout requests and use other connections to return the response messages simultaneously.

A correlation token is created by IMS to correlate the response message back to the correct IMS transaction instance. The capability of having different threads and connections for pulling callout requests and for returning response messages provides maximum concurrency.

The following high-level steps provide an overview of implementing and deploying your synchronous callout application and function.

1. Create or modify an IMS application for the ICAL call.
2. Define the OTMA destination descriptor for one of the following callout request approaches:
   - Send the callout request to an external application viaIMS Enterprise SuiteIMS TM Resource Adapter.
   - Send the callout request to an external application viaIMS Enterprise Suite SOAP Gateway.
   - Send the callout request to a user-written IMS Connect client application.

   The destination descriptor can be defined in the DFSYDTx member of the IMS.PROCLIB data set, or with the CREATE OTMADESC command.
3. Restart IMS for the newly defined OTMA descriptor. This step is not required if the descriptor was dynamically added with the CREATE OTMADESC command.
4. Run the IMS application that was created or modified in step 1 to issue the synchronous callout request.

Example COBOL program implementation of the synchronous callout function

To issue the ICAL call in a COBOL program, use the CALL statement.

```
CALL 'AIBTDLI' USING ICAL, AIB, CA-REQUEST, SCA-RESPONSE.
```

The following example demonstrates the required AIB field declaration for the ICAL call in the COBOL program. A complete COBOL example (with part name DFSSSCBL) is provided with the callout IVP sample in the SDFSSMPL sample library.

```
01 AIB.
   02 AIBRID PIC x(8)  VALUE 'DFSAIB  '.
   02 AIBRLEN PIC 9(9) USAGE BINARY.
   02 AIBSFUNC PIC x(8) VALUE 'SENDRECV'.
   02 AIBRSNM1 PIC x(8) VALUE 'OTMDEST1'.
   02 AIBRSNM2 PIC x(8).
   02 AIBRESV1 PIC x(8).
   02 AIBOALEN PIC 9(9) USAGE BINARY VALUE 28.
   02 AIBOAUSE PIC 9(9) USAGE BINARY VALUE 30.
   02 AIBRSFLD PIC 9(9) USAGE BINARY VALUE 5000.
   02 AIBRESV2 PIC x(8).
   02 AIBRETRN PIC 9(9) USAGE BINARY.
   02 AIBREASN PIC 9(9) USAGE BINARY.
   02 AIBERRXT PIC 9(9) USAGE BINARY.
 ...
```

The following example shows the CA-REQUEST and SCA-RESPONSE declarations in the COBOL program.

```
* ICAL Request Area
 01  CA-REQUEST.
     02  CA-MESSAGE   PIC  X(45) VALUE SPACES.

* ICAL Response Area
 01  SCA-RESPONSE.
     02  SCA-MESSAGE  PIC  X(100) VALUE SPACES.
```

Related concepts:

OTMA destination descriptors (Communications and Connections)

Related tasks:

Modifying an IMS application for callout requests

"Issuing synchronous callout requests from a Java dependent region" on page 660

Related reference:

ICAL call (Application Programming APIs)

Examples of DL/I call functions (Application Programming APIs)

Callout programming models

Implementing the asynchronous callout function

To issue an asynchronous callout request from your IMS application, issue the ISRT ALTPCB call and specify the OTMA destination descriptor name.

Any response to the callout request that is returned to IMS is handled as a new incoming transaction. If there is a response, the requesting application or a different IMS application must be coded to handle the response that is returned in a separate transaction.

Unlike synchronous callout requests, asynchronous callout requests do not require the IMS application program that issues the request to wait for a response in the dependent region. After it issues an asynchronous callout request, the application program can terminate and free the dependent region. Any response to the callout request that is returned to IMS is handled as a new incoming transaction and IMS schedules a new application program instance to process it.

If an asynchronous callout request generates a response, however, the benefit gained by freeing dependent regions might be offset by the additional complexity of managing the response. For asynchronous callout responses, your installation is responsible for developing the method for correlating the response to the original request. For synchronous callout requests, IMS manages that correlation.

The following high-level steps provide an overview of implementing and deploying your asynchronous callout application and function.

1. Plan for the correlation of asynchronous callout responses.
2. Create or modify an IMS application to issue an ISRT ALTPCB call for asynchronous callout requests.
3. Define the callout routing information. There are two options to define the required information:
   - Define an OTMA routing descriptor.
   - Code the DFSYPRX0 and DFSYDRU0 exit routines.
4. Optional: Restart IMS for the newly defined OTMA descriptor. A restart is required only if you create or modify an OTMA routing descriptor in the

DFSYDTx member of the IMS.PROCLIB data set. You do not need to restart IMS if you use the CREATE OTMADESC or UPDATE OTMADESC commands.

5. Run the IMS application that issues the callout request. The IMS application is usually triggered through an initiating client, such as a terminal, or an IMS Connect or OTMA client.

Related concepts:

Asynchronous callout request (Communications and Connections)

OTMA destination descriptors (Communications and Connections)

Related reference:

OTMA User Data Formatting exit routine (DFSYDRU0) (Exit Routines)

OTMA Destination Resolution exit routine (DFSYPRX0) (Exit Routines)

Part 4. Application programming for EXEC DLI

IMS provides support for writing applications to access IMS resources using EXEC DLI.

© Copyright IBM Corp. 1974, 2015

Chapter 28. Writing your application programs for EXEC DLI

You can write programs in assembler language, COBOL, PL/I, C, and C++ that execute EXEC DLI commands to access IMS.

Programming guidelines

Use the following guidelines to write efficient and error-free EXEC DL/I programs.

The number, type, and sequence of the DL/I requests your program issues affect the efficiency of your program. A program that is poorly designed runs if it is coded correctly. The suggestions that follow can help you develop the most efficient design possible for your application program. Inefficiently designed programs can adversely affect performance and are hard to change. Being aware of how certain combinations of commands or calls affects performance helps you to avoid these problems and design a more efficient program.

After you have a general sequence of calls mapped out for your program, use these guidelines to improve the sequence. Usually an efficient sequence of requests causes efficient internal DL/I processing.

- Use the simplest call. Qualify your requests to narrow the search for DL/I, but do not use more qualification than required.
- Use the request or sequence of requests that gives DL/I the shortest path to the segment you want.
- Use the fewest number of requests possible in your program. Each DL/I request your program issues uses system time and resources. You may be able to eliminate unnecessary calls by:
  - Using path requests if you are replacing, retrieving, or inserting more than one segment in the same path. If you are using more than one request to do this, you are issuing unnecessary requests.
  - Changing the sequence so that your program saves the segment in a separate I/O area, and then gets it from that I/O area the second time it needs the segment. If your program retrieves the same segment more than once during program execution, you are issuing an unnecessary request.
  - Anticipating and eliminating needless and nonproductive requests, such as requests that result in GB, GE, and II status codes. For example, if you are issuing `GN`s for a particular segment type and you know how many occurrences of that segment type exist, do not issue the `GN` that results in a GE status code. You can keep track of the number of occurrences your program retrieves, and then continue with other processing when you know you have retrieved all the occurrences of that segment type.
  - Issuing an insert request with a qualification for each parent instead of issuing Get requests for the parents to make sure that they exist. When you are inserting segments, you cannot insert dependents unless the parents exist. If DL/I returns a GE status code, at least one of the parents does not exist.
- Keep the main section of the program logic together. For example, branch to conditional routines, such as error and print routines, in other parts of the program, instead of having to branch around them to continue normal processing.

© Copyright IBM Corp. 1974, 2015

- Use call sequences that make good use of the physical placement of the data. Access segments in hierarchical sequence as much as possible. Avoid moving backward in the hierarchy.
- Process database records in order of the key field of the root segments. (For HDAM databases, this order depends on the randomizing routine that is used. Check with your DBA for this information.)
- Try to avoid constructing the logic of the program and the structure of commands or calls in a way that depends heavily on the database structure. Depending on the current structure of the hierarchy reduces the program's flexibility.

Coding a program in assembler language

The following sample assembler language program shows how the different parts of a command-level program fit together, and how the EXEC DLI commands are coded in a CICS online program.

Except for a few commands, this program applies to batch, BMP, and CICS programs. Any differences are highlighted in the notes for the sample assembler code. The numbering on the right of the sample code references these notes.

```
*ASM XOPTS(CICS,DLI)
*                                                                    1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R11      EQU   11
R12      EQU   12
R13      EQU   13
DFHEISTG DSECT
SEGKEYA  DS    CL4
SEGKEYB  DS    CL4                                                   2
SEGKEYC  DS    CL4
SEGKEY1  DS    CL4
SEGKEY2  DS    CL4
CONKEYB  DS    CL8
SEGNAME  DS    CL8
SEGLEN   DS    H
PCBNUM   DS    H
AREAA    DS    CL80
AREAB    DS    CL80                                                  3
AREAC    DS    CL80
AREAG    DS    CL250
AREASTAT DS    CL360
*        COPY  MAPSET
*
*****************************************************************
*  INITIALIZATION
*  HANDLE ERROR CONDITIONS IN ERROR ROUTINE                          4
*  HANDLE ABENDS (DLI ERROR STATUS CODES) IN ABEND ROUTINE
*  RECEIVE INPUT MESSAGE
*****************************************************************
*
SAMPLE   DFHEIENT CODEREG=(R2,R3),DATAREG=(R13,R12),EIBREG=R11         5
*
         EXEC CICS HANDLE CONDITION ERROR(ERRORS)                      6
*
         EXEC CICS HANDLE ABEND LABEL(ABENDS)                          6
*
         EXEC CICS RECEIVE MAP ('SAMPMAP') MAPSET('MAPSET')            6
*        ANALYZE INPUT MESSAGE AND PERFORM NON-DLI PROCESSING
*
*****************************************************************
*  SCHEDULE PSB NAMED 'SAMPLE1'
```

```
*****************************************************************
*
         EXEC DLI SCHD PSB(SAMPLE1)                                    7
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  RETRIEVE ROOT SEGMENT AND ALL ITS DEPENDENTS
*****************************************************************
*
         MVC   SEGKEYA,=C'A300'                                        8
         EXEC DLI GU USING PCB(1) SEGMENT(SEGA) INTO(AREAA)           X
               SEGLENGTH(80) WHERE(KEYA=SEGKEYA) FIELDLENGTH(4)
         BAL   R4,TESTDIB          CHECK STATUS
GNPLOOP  EQU   *
         EXEC DLI GNP USING PCB(1) INTO(AREAG) SEGLENGTH(250)
         CLC   DIBSTAT,=C'GE'      LOOK FOR END                        9
         BE    LOOPDONE            DONE AT 'GE'
         BAL   R4,TESTDIB          CHECK STATUS
         B     GNPLOOP
LOOPDONE EQU   *
*
*****************************************************************
*  INSERT NEW ROOT SEGMENT
*****************************************************************
*
         MVC   AREAA,=CL80'DATA FOR NEW SEGMENT INCLUDING KEY'
         EXEC DLI ISRT USING PCB(1) SEGMENT(SEGA) FROM(AREAA)         X
               SEGLENGTH(80)
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  RETRIEVE 3 SEGMENTS IN PATH AND REPLACE THEM
*****************************************************************
*
         MVC   SEGKEYA,=C'A200'
         MVC   SEGKEYB,=C'B240'
         MVC   SEGKEYC,=C'C241'
         EXEC DLI GU USING PCB(1)                                     X
               SEGMENT(SEGA) WHERE(KEYA=SEGKEYA)                      X 10
               FIELDLENGTH(4)                                         X
               INTO(AREAA)                                            X
               SEGLENGTH(80)                                          X
               SEGMENT(SEGB) WHERE(KEYB=SEGKEYB) FIELDLENGTH(4)       X
               INTO(AREAB)                                            X
               SEGLENGTH(80)                                          X
               SEGMENT(SEGC) WHERE(KEYC=SEGKEYC) FIELDLENGTH(4)       X
               INTO(AREAC)                                            X
               SEGLENGTH(80)
         BAL   R4,TESTDIB
*        UPDATE FIELDS IN THE 3 SEGMENTS
         EXEC DLI REPL USING PCB(1)                                   X
               SEGMENT(SEGA) FROM(AREAA) SEGLENGTH(80)                X
               SEGMENT(SEGB) FROM(AREAB) SEGLENGTH(80)                X
               SEGMENT(SEGC) FROM(AREAC) SEGLENGTH(80)
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  INSERT NEW SEGMENT USING CONCATENATED KEY TO QUALIFY PARENT
*****************************************************************
*
         MVC   AREAC,=CL80'DATA FOR NEW SEGMENT INCLUDING KEY'
         MVC   CONKEYB,=C'A200B240'
         EXEC DLI ISRT USING PCB(1)                                   X
               SEGMENT(SEGB) KEYS(CONKEYB) KEYLENGTH(8)               X
               SEGMENT(SEGC) FROM(AREAC) SEGLENGTH(80)
         BAL   R4,TESTDIB          CHECK STATUS
*
```

```
*****************************************************************
*  RETRIEVE SEGMENT DIRECTLY USING CONCATENATED KEY
*  AND THEN DELETE IT AND ITS DEPENDENTS
*****************************************************************
*
         MVC   CONKEYB,=C'A200B230'
         EXEC DLI GU USING PCB(1)                                  X
               SEGMENT(SEGB)                                       X
               KEYS(CONKEYB) KEYLENGTH(8)                          X
               INTO(AREAB) SEGLENGTH(80)
         BAL   R4,TESTDIB          CHECK STATUS
         EXEC DLI DLET USING PCB(1)                                X
               SEGMENT(SEGB) SEGLENGTH(80) FROM(AREAB)
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  RETRIEVE SEGMENT BY QUALIFYING PARENT WITH CONCATENATED KEY,
*  OBJECT SEGMENT WITH WHERE OPTION USING A LITERAL,
*  AND THEN SET PARENTAGE
*
*  USE VARIABLES FOR PCB INDEX, SEGMENT NAME, AND SEGMENT LENGTH
*****************************************************************
*
         MVC   CONKEYB,=C'A200B230'
         MVC   SEGNAME,=CL8'SEGA'
         MVC   SEGLEN,=H'80'
         MVC   PCBNUM,=H'1'
         EXEC DLI GU USING PCB(PCBNUM)                             X
               SEGMENT((SEGNAME))                                  X
               KEYS(CONKEYB) KEYLENGTH(8) SETPARENT                X
               SEGMENT(SEGC) INTO(AREAC) SEGLENGTH(SEGLEN)         X
               WHERE(KEYC='C520')
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  RETRIEVE DATABASE STATISTICS
*****************************************************************
*
         EXEC DLI STAT USING PCB(1) INTO(AREASTAT)                 X
               VSAM FORMATTED LENGTH(360)
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  RETRIEVE ROOT SEGMENT USING BOOLEAN OPERATORS
*****************************************************************
*
         MVC   SEGKEY1,=C'A050'
         MVC   SEGKEY2,=C'A150'
         EXEC DLI GU USING PCB(1) SEGMENT(SEGA) INTO(AREAA)        X
               SEGLENGTH(80) FIELDLENGTH(4,4,4,4)                  X
               WHERE(KEYA > SEGKEY1 AND KEYA < SEGKEY2
               KEYA > 'A275' AND  KEYA < 'A350')
         BAL   R4,TESTDIB          CHECK STATUS
*
*****************************************************************
*  TERMINATE PSB WHEN DLI PROCESSING IS COMPLETED
*****************************************************************
*
         EXEC DLI TERM                                            11
*
*****************************************************************
*  SEND OUTPUT MESSAGE
*****************************************************************
*
         EXEC CICS SEND MAP('SAMPMAP') MAPSET('MAPSET')            6
         EXEC CICS WAIT TERMINAL
*
```

```
*****************************************************************
*  COMPLETE TRANSACTION AND RETURN TO CICS
*****************************************************************
*
         EXEC CICS RETURN                                          12
*
*****************************************************************
*  CHECK STATUS IN DIB
*****************************************************************
*
TESTDIB  EQU   *
         CLC   DIBSTAT,=C'  '      IS STATUS BLANK                 13
         BER   R4                  YES - RETURN
*        HANDLE DLI STATUS CODES REPRESENTING EXCEPTIONAL CONDITIONS
*
         BR    R4                  RETURN
ERRORS   EQU   *
*        HANDLE ERROR CONDITIONS
*
ABENDS   EQU   *
*        HANDLE ABENDS INCLUDING DLI ERROR STATUS CODES
*
         END
```

Notes for the sample assembler code:

1 For a CICS online program containing EXEC DLI commands, you must specify the DLI and CICS options. For a batch or BMP program containing EXEC DLI, you must specify only the DLI option.

2 For reentry, define each of the areas the program uses—I/O areas, key feedback areas, and segment name areas in DFHEISTG.

3 Define an I/O area for each segment you retrieve, add, or replace (in a single command).

4 For a batch or BMP program containing EXEC DLI, you must save registers on entry and restore registers on exit according to z/OS register-saving conventions.

5 In a batch or BMP program, aDFHEIRET with an optional DFHEIENT saves the registers on entry. Do not specify the EIBREG parameter in a batch program.

6 Do not code EXEC CICS commands in a batch or BMP program.

7 In a CICS online program, use the `SCHD PSB` command to obtain a PSB for the use of your program. Do *not* schedule a PSB in a batch or BMP program.

8 This `GU` command retrieves the first occurrence of SEGA with a key of A300. You do not have to provide the KEYLENGTH or SEGLENGTH options in an assembler language program.

9 This `GNP` command retrieves all dependents under segment SEGA. The GE status code indicates that no more dependents exist.

10 This `GU` command is an example of a path command. Use a separate I/O area for each segment you retrieve.

11 In a CICS online program, the `TERM` command terminates the PSB scheduled earlier. You do *not* terminate the PSB in a batch or BMP program.

12 For a batch or BMP program, code RCREG parameter instead of EXEC CICS RETURN. The RCREG parameter identifies a register containing the return code.

13 After issuing each command, you should check the status code in the DIB.

Coding a program in COBOL

The following sample COBOL program shows how the different parts of a command-level program fit together, and how the EXEC DLI commands are coded in a CICS online program.

Except for a few commands, this program applies to batch, BMP, and CICS programs. Any differences are highlighted in the notes for the sample COBOL code. The numbering on the right of the sample code references the notes.

```
CBL LIB,APOST,XOPTS(CICS,DLI)        IDENTIFICATION DIVISION.
       PROGRAM-ID. SAMPLE.                                          1
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
.*     SOURCE-COMPUTER. IBM-370.
.*     OBJECT-COMPUTER. IBM-370.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77  SEGKEYA               PIC X(4).
       77  SEGKEYB               PIC X(4).                          2
       77  SEGKEYC               PIC X(4).
       77  SEGKEY1               PIC X(4).
       77  SEGKEY2               PIC X(4).
       77  SEGKEY3               PIC X(4).
       77  SEGKEY4               PIC X(4).
       77  CONKEYB               PIC X(8).
       77  SEGNAME               PIC X(8).
       77  SEGLEN                COMP PIC S9(4).
       77  PCBNUM                COMP PIC S9(4).
       01  AREAA                 PIC X(80).
      *    DEFINE SEGMENT I/O AREA
       01  AREAB                 PIC X(80).
       01  AREAC                 PIC X(80).                         3
       01  AREAG                 PIC X(250).
       01  AREASTAT              PIC X(360).
      *    COPY MAPSET.
        PROCEDURE DIVISION.
      *
      * ***************************************************************
      *  INITIALIZATION
      *  HANDLE ERROR CONDITIONS IN ERROR ROUTINE
      *  HANDLE ABENDS (DLI ERROR STATUS CODES) IN ABEND ROUTINE
      *  RECEIVE INPUT MESSAGE
      * ***************************************************************
      *
           EXEC CICS HANDLE CONDITION ERROR(ERRORS) END-EXEC.         4
      *
           EXEC CICS HANDLE ABEND LABEL(ABENDS) END-EXEC.             4
      *
           EXEC CICS RECEIVE MAP ('SAMPMAP') MAPSET('MAPSET') END-EXEC. 4
      *    ANALYZE INPUT MESSAGE AND PERFORM NON-DLI PROCESSING
      *
      * ***************************************************************
      *  SCHEDULE PSB NAMED 'SAMPLE1'
      * ***************************************************************
      *
           EXEC DLI SCHD PSB(SAMPLE1) END-EXEC.
           PERFORM TEST-DIB THRU OK.                                  5
      *
      * ***************************************************************
      *  RETRIEVE ROOT SEGMENT AND ALL ITS DEPENDENTS
      * ***************************************************************
      *
           MOVE 'A300' TO SEGKEYA.
           EXEC DLI GU USING PCB(1) SEGMENT(SEGA) INTO(AREAA)
                SEGLENGTH(80) WHERE(KEYA=SEGKEYA)                     6
                FIELDLENGTH(4)
```

```
           END-EXEC.
           PERFORM TEST-DIB THRU OK.
       GNPLOOP.
           EXEC DLI GNP USING PCB(1) INTO(AREAG) SEGLENGTH(250)
           END-EXEC.
           IF DIBSTAT EQUAL TO 'GE' THEN GO TO LOOPDONE.
           PERFORM TEST-DIB THRU OK.
           GO TO GNPLOOP.
       LOOPDONE.
*
      * ***************************************************************
      *  INSERT NEW ROOT SEGMENT
      * ***************************************************************
      *
           MOVE 'DATA FOR NEW SEGMENT INCLUDING KEY' TO AREAA.
           EXEC DLI ISRT USING PCB(1) SEGMENT(SEGA) FROM(AREAA)
                SEGLENGTH(80) END-EXEC.
           PERFORM TEST-DIB THRU OK.
      *
      * ***************************************************************
      * RETRIEVE 3 SEGMENTS IN PATH AND REPLACE THEM
      * ***************************************************************
      *
           MOVE 'A200' TO SEGKEYA.
           MOVE 'B240' TO SEGKEYB.
           MOVE 'C241' TO SEGKEYC.
           EXEC DLI GU USING PCB(1)
             SEGMENT(SEGA) WHERE(KEYA=SEGKEYA) FIELDLENGTH(4)          7
                INTO(AREAA)
                SEGLENGTH(80)
             SEGMENT(SEGB) WHERE(KEYB=SEGKEYB) FIELDLENGTH(4)
                INTO(AREAB)
                SEGLENGTH(80)
             SEGMENT(SEGC) WHERE(KEYC=SEGKEYC) FIELDLENGTH(4)
                INTO(AREAC)
                SEGLENGTH(80)
           END-EXEC.
           PERFORM TEST-DIB THRU OK.
      *    UPDATE FIELDS IN THE 3 SEGMENTS
           EXEC DLI REPL USING PCB(1)
             SEGMENT(SEGA) FROM(AREAA) SEGLENGTH(80)
             SEGMENT(SEGB) FROM(AREAB) SEGLENGTH(80)
             SEGMENT(SEGC) FROM(AREAC) SEGLENGTH(80)
           END-EXEC.
           PERFORM TEST-DIB THRU OK.
      *
      * ***************************************************************
      *  INSERT NEW SEGMENT USING CONCATENATED KEY TO QUALIFY PARENT
      * ***************************************************************
      *
           MOVE 'DATA FOR NEW SEGMENT INCLUDING KEY' TO AREAC.
           MOVE 'A200B240' TO CONKEYB.
           EXEC DLI ISRT USING PCB(1)
             SEGMENT(SEGB) KEYS(CONKEYB) KEYLENGTH(8)
             SEGMENT(SEGC) FROM(AREAC) SEGLENGTH(80)
           END-EXEC.
           PERFORM TEST-DIB THRU OK.
      *
      * ***************************************************************
      *  RETRIEVE SEGMENT DIRECTLY USING CONCATENATED KEY
      *  AND THEN DELETE IT AND ITS DEPENDENTS
      * ***************************************************************
      *
           MOVE 'A200B230' TO CONKEYB.
           EXEC DLI GU USING PCB(1)
             SEGMENT(SEGB)
                KEYS(CONKEYB) KEYLENGTH(8)
```

```
           INTO(AREAB) SEGLENGTH(80)
         END-EXEC.
         PERFORM TEST-DIB THRU OK.
         EXEC DLI DLET USING PCB(1)
           SEGMENT(SEGB) SEGLENGTH(80) FROM(AREAB) END-EXEC.
         PERFORM TEST-DIB THRU OK.
     *
* ************************************************************
     *  RETRIEVE SEGMENT BY QUALIFYING PARENT WITH CONCATENATED KEY,
     *  OBJECT SEGMENT WITH WHERE OPTION,
     *  AND THEN SET PARENTAGE
     *
     *  USE VARIABLES FOR PCB INDEX, SEGMENT NAME, AND SEGMENT LENGTH
     * ************************************************************
     *
          MOVE 'A200B230' TO CONKEYB.
          MOVE 'C520' TO SEGKEYC.
          MOVE 'SEGA' TO SEGNAME.
          MOVE 80 TO SEGLEN.
          MOVE 1 TO PCBNUM.
          EXEC DLI GU USING PCB(PCBNUM)
            SEGMENT((SEGNAME))
               KEYS(CONKEYB) KEYLENGTH(8) SETPARENT
            SEGMENT(SEGC) INTO(AREAC) SEGLENGTH(SEGLEN)
               WHERE(KEYC=SEGKEYC) FIELDLENGTH(4) END-EXEC.
          PERFORM TEST-DIB THRU OK.
     *
     * ************************************************************
     *  RETRIEVE DATABASE STATISTICS
     * ************************************************************
     *
          EXEC DLI STAT USING PCB(1) INTO(AREASTAT)
               VSAM FORMATTED LENGTH(360) END-EXEC.
          PERFORM TEST-DIB THRU OK.
     *
     * ************************************************************
     *  RETRIEVE ROOT SEGMENT USING BOOLEAN OPERATORS
     * ************************************************************
     *
          MOVE 'A050' TO SEGKEY1.
          MOVE 'A150' TO SEGKEY2.
          MOVE 'A275' TO SEGKEY3.
          MOVE 'A350' TO SEGKEY4.
          EXEC DLI GU USING PCB(1) SEGMENT(SEGA) INTO(AREAA)
               SEGLENGTH(80) FIELDLENGTH(4,4,4,4)
               WHERE(KEYA > SEGKEY1 AND KEYA < SEGKEY2 OR
                     KEYA > SEGKEY3 AND KEYA < SEGKEY4)
          END-EXEC.
          PERFORM TEST-DIB THRU OK.
     *
  * ************************************************************
     *  TERMINATE PSB WHEN DLI PROCESSING IS COMPLETED
     * ************************************************************
     *
            EXEC DLI TERM END-EXEC.                                  8
     *
     * ************************************************************
     * ************************************************************
     *  SEND OUTPUT MESSAGE
     * ************************************************************
     *
          EXEC CICS SEND MAP('SAMPMAP') MAPSET('MAPSET') END-EXEC.
          EXEC CICS WAIT TERMINAL END-EXEC.
     *
     * ************************************************************
     *  COMPLETE TRANSACTION AND RETURN TO CICS
     * ************************************************************
```

```
*
     EXEC CICS RETURN END-EXEC.
*
* ************************************************************
*  CHECK STATUS IN DIB
* ************************************************************
*
 TEST-DIB.
     IF DIBSTAT EQUAL TO '  ' THEN GO TO OK.
 OK.                                                          9
 ERRORS.
*    HANDLE ERROR CONDITIONS
 ABENDS.
*    HANDLE ABENDS INCLUDING DLI ERROR STATUS CODES
```

Notes for the sample COBOL code:

1 For a CICS online program containing EXEC DLI commands, you must specify the DLI and CICS options. For a batch or BMP program containing EXEC DLI, you must specify only the DLI option.

2 Define each of the areas the program uses—I/O areas, key feedback areas, and segment name areas—as 77- or 01-level working storage entries.

3 Define an I/O area for each segment you retrieve, add, or replace (in a single command).

4 Do not code EXEC CICS commands in a batch or BMP program.

5 For CICS online programs, you use a `SCHD PSB` command to obtain a PSB. You do *not* schedule a PSB in a batch or BMP program.

6 This `GU` command retrieves the first occurrence of SEGA with a key of A300. KEYLENGTH and SEGLENGTH are optional for IBM COBOL for z/OS & VM (and VS COBOL II). For COBOL V4 and OS/VS COBOL, KEYLENGTH and SEGLENGTH are required.

7 This `GU` command is an example of a path command. You must use a separate I/O area for each segment you retrieve.

8 For a CICS online program, the `TERM` command terminates the PSB scheduled earlier. You do *not* terminate the PSB in a batch or BMP program.

9 After issuing each command, you should check the status code in the DIB.

Coding a program in PL/I

The following sample PL/I program shows how the different parts of a command-level program fit together, and how the EXEC DLI commands are coded in a CICS online program.

Except for a few commands, this program applies to batch, BMP, and CICS programs. Any differences are highlighted in the notes for the sample PL/I code. The numbering on the right of the sample code references those notes.

```
*PROCESS INCLUDE,GN,XOPTS(CICS,DLI);                          1
 SAMPLE: PROCEDURE OPTIONS(MAIN);
 DCL      SEGKEYA                 CHAR (4);
 DCL      SEGKEYB                 CHAR (4);                    2
 DCL      SEGKEYC                 CHAR (4);
 DCL      SEGKEY1                 CHAR (4);
 DCL      SEGKEY2                 CHAR (4);
 DCL      SEGKEY3                 CHAR (4);
 DCL      SEGKEY4                 CHAR (4);
 DCL      CONKEYB                 CHAR (8);
 DCL      SEGNAME                 CHAR (8);
 DCL      PCBNUM                  FIXED BIN (15);
 DCL      AREAA                   CHAR (80);
```

```
    /*   DEFINE SEGMENT I/O AREA                                    */
 DCL     AREAB                  CHAR (80);
 DCL     AREAC                  CHAR (80);                                3
 DCL     AREAG                  CHAR (250);
 DCL     AREASTAT               CHAR (360);
    %INCLUDE MAPSET
    /*                                                              */
    /*                                                              */
    /* ************************************************************ */
    /*  INITIALIZATION                                              */
    /*  HANDLE ERROR CONDITIONS IN ERROR ROUTINE                    */
    /*  HANDLE ABENDS (DLI ERROR STATUS CODES) IN ABEND PROGRAM     */
    /*  RECEIVE INPUT MESSAGE                                       */
    /* ************************************************************ */
    /*                                                              */
    EXEC CICS HANDLE CONDITION ERROR(ERRORS);                        4
    /*                                                              */
    EXEC CICS HANDLE ABEND PROGRAM('ABENDS');                        4
    /*                                                              */
    EXEC CICS RECEIVE MAP ('SAMPMAP') MAPSET('MAPSET');              4
    /* ANALYZE INPUT MESSAGE AND PERFORM NON-DLI PROCESSING         */
    /*                                                              */
    /* ************************************************************ */
    /*  SCHEDULE PSB NAMED 'SAMPLE1'                                */
    /* ************************************************************ */
    /*                                                              */
    EXEC DLI SCHD PSB(SAMPLE1);
    CALL TEST_DIB;                                                   5

    /* ************************************************************ */
    /*  RETRIEVE ROOT SEGMENT AND ALL ITS DEPENDENTS                */
    /* ************************************************************ */
    /*                                                              */
    SEGKEYA = 'A300';
    EXEC DLI GU USING PCB(1) SEGMENT(SEGA) INTO(AREAA)
    WHERE(KEYA=SEGKEYA);                                             6
    CALL TEST_DIB;
GNPLOOP:
    EXEC DLI GNP USING PCB(1) INTO(AREAG);                           7
    IF DIBSTAT = 'GE' THEN GO TO LOOPDONE;
    CALL TEST_DIB;
    GO TO GNPLOOP;
 LOOPDONE:
    /*                                                              */
    /* ************************************************************ */
    /*  INSERT NEW ROOT SEGMENT                                     */
    /* ************************************************************ */
    /*                                                              */
    AREAA = 'DATA FOR NEW SEGMENT INCLUDING KEY';
    EXEC DLI ISRT USING PCB(1) SEGMENT(SEGA) FROM(AREAA);
    CALL TEST_DIB;
    /*                                                              */
    /* ************************************************************ */
    /*  RETRIEVE 3 SEGMENTS IN PATH AND REPLACE THEM                */
    /* ************************************************************ */
    /*                                                              */
    SEGKEYA = 'A200';
    SEGKEYB = 'B240';
    SEGKEYC = 'C241';
    EXEC DLI GU USING PCB(1)
      SEGMENT(SEGA) WHERE(KEYA=SEGKEYA)                              8
         INTO(AREAA)
      SEGMENT(SEGB) WHERE(KEYB=SEGKEYB)
         INTO(AREAB)
      SEGMENT(SEGC) WHERE(KEYC=SEGKEYC)
         INTO(AREAC);
    CALL TEST_DIB;
```

```
    /* UPDATE FIELDS IN THE 3 SEGMENTS                              */
    EXEC DLI REPL USING PCB(1)
      SEGMENT(SEGA) FROM(AREAA)
      SEGMENT(SEGB) FROM(AREAB)
      SEGMENT(SEGC) FROM(AREAC);
    CALL TEST_DIB;
    /*                                                              */
    /* ************************************************************ */
    /*  INSERT NEW SEGMENT USING CONCATENATED KEY TO QUALIFY PARENT  */
    /* ************************************************************ */
    /*                                                              */
    AREAC = 'DATA FOR NEW SEGMENT INCLUDING KEY';
    CONKEYB = 'A200B240';
    EXEC DLI ISRT USING PCB(1)
      SEGMENT(SEGB) KEYS(CONKEYB)
      SEGMENT(SEGC) FROM(AREAC);
    CALL TEST_DIB;
    /*                                                              */
    /* ************************************************************  */
    /*  RETRIEVE SEGMENT DIRECTLY USING CONCATENATED KEY            */
    /*  AND THEN DELETE IT AND ITS DEPENDENTS                       */
    /* ************************************************************  */
    /*                                                              */
    CONKEYB = 'A200B230';
    EXEC DLI GU USING PCB(1)
      SEGMENT(SEGB)
         KEYS(CONKEYB)
         INTO(AREAB);
    CALL TEST_DIB;
    EXEC DLI DLET USING PCB(1)
      SEGMENT(SEGB) FROM(AREAB);
    CALL TEST_DIB;
    /*                                                              */
/* ************************************************************ */
    /*  RETRIEVE SEGMENT BY QUALIFYING PARENT WITH CONCATENATED KEY, */
    /*  OBJECT SEGMENT WITH WHERE OPTION                            */
    /*  AND THEN SET PARENTAGE                                      */
    /*                                                              */
    /*  USE VARIABLES FOR PCB INDEX, SEGMENT NAME                   */
    /* ************************************************************ */
    /*                                                              */
    CONKEYB = 'A200B230';
    SEGNAME = 'SEGA';
    SEGKEYC = 'C520';
    PCBNUM = 1;
    EXEC DLI GU USING PCB(PCBNUM)
      SEGMENT((SEGNAME))
         KEYS(CONKEYB) SETPARENT
      SEGMENT(SEGC) INTO(AREAC)
         WHERE(KEYC=SEGKEYC);
    CALL TEST_DIB;
    /*                                                              */
    /* ************************************************************ */
    /*  RETRIEVE DATABASE STATISTICS                                */
    /* ************************************************************ */
    /*                                                              */
    EXEC DLI STAT USING PCB(1) INTO(AREASTAT) VSAM FORMATTED;
    CALL TEST_DIB;
    /*                                                              */
    /* ************************************************************  */
    /*  RETRIEVE ROOT SEGMENT USING BOOLEAN OPERATORS               */
    /* ************************************************************  */
    /*                                                              */
    SEGKEY1 = 'A050';
    SEGKEY2 = 'A150';
    SEGKEY3 = 'A275';
    SEGKEY4 = 'A350';
```

```
    EXEC DLI GU USING PCB(1) SEGMENT(SEGA) INTO(AREAA)
         WHERE(KEYA &Ar; SEGKEY1 AND KEYA &Al; SEGKEY2 OR
               KEYA &Ar; SEGKEY3 AND KEYA &Al; SEGKEY4);
    CALL TEST_DIB;
    /*                                                            */
    /* ********************************************************** */
    /*  TERMINATE PSB WHEN DLI PROCESSING IS COMPLETED            */
    /* ********************************************************** */
    /*                                                            */

    EXEC DLI TERM;                                                 9

    /*                                                            */
    /* ********************************************************** */
    /*  SEND OUTPUT MESSAGE                                       */
    /* ********************************************************** */
    /*                                                            */
    EXEC CICS SEND MAP('SAMPMAP') MAPSET('MAPSET');                4
    EXEC CICS WAIT TERMINAL;
    /*                                                            */
    /* ********************************************************** */
    /*  COMPLETE TRANSACTION AND RETURN TO CICS                   */
    /* ********************************************************** */
    /*                                                            */
    EXEC CICS RETURN;                                              4
    /*                                                            */
    /* ********************************************************** */
    /*  CHECK STATUS IN DIB                                       */
    /* ********************************************************** */
    /*                                                            */
TEST_DIB: PROCEDURE;
    IF DIBSTAT = '  ' RETURN;                                     10

    /* HANDLE DLI STATUS CODES REPRESENTING EXCEPTIONAL CONDITIONS */
    /*                                                            */
OK:
END TEST_DB;
ERRORS:
    /* HANDLE ERROR CONDITIONS                                   */
    /*                                                            */
END SAMPLE;
```

Notes to the sample PL/I code:

1 For a CICS online program containing EXEC DLI commands, you must specify the DLI and CICS options. For a batch or BMP program containing EXEC DLI, you must specify only the DLI option.

2 Define, in automatic storage, each of the areas; I/O areas, key feedback areas, and segment name areas.

3 Define an I/O area for each segment you retrieve, add, or replace in a single command.

4 Do not code EXEC CICS commands in a batch or BMP program.

5 For CICS online programs, you use a `SCHD PSB` command to obtain a PSB. You do *not* schedule a PSB in a batch or BMP program.

6 This `GU` command retrieves the first occurrence of SEGA with a key of A300. Notice that you do not need to include the KEYLENGTH and SEGLENGTH options.

7 This `GNP` command retrieves all dependents under segment SEGA. TheGE status code indicates that no more dependents exist.

8 This `GU` command is an example of a path command. You must use a separate I/O area for each segment you retrieve.

9 For a CICS online program, the TERM command terminates the PSB scheduled earlier. You do *not* terminate the PSB in a batch or BMP program.

10 After issuing each command, you should check the status code in the DIB.

Coding a program in C

the following sample C program shows how the different parts of a command-level program fit together, and how the EXEC DLI commands are coded in a CICS online program.

Except for a few commands, this program applies to batch, BMP, and CICS programs. Any differences are highlighted in the notes for the sample C code. The numbering on the right of the sample code references those notes.

```
#include < string.h>                                                    1
#include < stdio.h >                                                    2

   char  DIVIDER[120] = "----------------------------------------\
 ------------------------------------------------------------";
   char  BLANK[120]   = "                                              \
                                                                       \0";
   char  BLAN2[110]   = "                                              \
                                                 \0";
   char  SCHED[120] = "Schedule PSB(PC3COCHD)         "                          3
   char  GN1[120] = "GN using PCB(2) Segment(SE2ORDER) check dibstat \
is blank";
   char   GNP1[120] = "GNP using PCB(2) check dibstat = GK or blank \
(or GE for last GNP)";
   char   GU1[120]  = "GU using PCB(2) Segment(SE2ORDER) where(\
FE2OGREF=000000'') check dibstat blank";
   char   GU2[120]  = "GU using PCB(2) Segment(SE2ORDER) where(\
FE2OGREF=000999'') check dibstat blank";
   char  REP1[120]  = "REPLACE using PCB(2) Segment(SE2ORDER) check \
dibstat is blank";
   char  DEL1[120]   = "DELETE using PCB(2) Segment(SE2ORDER) check \
dibstat is blank";
   char  INS1[120]   = "INSERT using PCB(2) Segment(SE2ORDER) where\
(FE2OGREF=''000999'') check dibstat is blank";
   char  TERM[120]   = "TERM - check dibstat is blank";
   char  STAT[120]   = "STAT USING PCB(2) VSAM FORMATTED";
   char  DATAB[6]    = "000999";
   char  DATAC[114]  = " REGRUN TEST INSERT NO1.";
   char  START[120]  = "PROGXIV STARTING";
   char  OKMSG[120]  = "PROGXIV COMPLETE";
   int   TLINE = 120;
   int   L11 = 11;
   int   L360 = 11;
   struct {
         char NEWSEGB[6];
         char NEWSEGC[54];
   } NEWSEG;
   char  OUTLINE[120];                                                      4
   struct {
         char OUTLINA[9];
         char OUTLINB[111];
   } OUTLIN2;
   struct {
         char  OUTLINX[9];
         char  OUTLINY[6];
         char  OUTLINZ[105];
   } OUTLIN3;
   char  GUIOA[60];
   char  GNIOA[60];
   struct {
         char ISRT1[6];
         char ISRT2[54];
```

```
   } ISRTIOA;
   struct {
         char REPLIO1[6];
         char REPLIO2[54];
   } REPLIOA;
   struct {
         char DLET1[6];
         char DLET2[54];
   } DLETIOA;
   struct {
         char STATA1[120];
         char STATA2[120];
         char STATA3[120];
   } STATAREA;
   struct {
         char DHPART[2];
         char RETCODE[2]
   } DHABCODE;

main()
{
         EXEC CICS ADDRESS EIB(dfheiptr);                          5
         strcpy(OUTLINE,DIVIDER);
         SENDLINE();
         strcpy(OUTLINE,START);
         SENDLINE();
    /*                                                          */
    /* SCHEDULE PSB                                             */
    /*                                                          */
         strcpy(OUTLINE,SCHED);
         SENDLINE();
         EXEC DLI SCHEDULE PSB(PC3COCHD);                          6
         SENDSTAT();
         TESTDIB();
    /*                                                         */
    /* ISSUE GU REQUEST                                        */
    /*                                                         */
         strcpy(OUTLINE,GU1);
         SENDLINE();
         EXEC DLI GET UNIQUE USING PCB(2)                          7
               SEGMENT(SE2ORDER)
               WHERE(FE2OGREF>="000000")
               INTO(&GUIOA) SEGLENGTH(60);
         strcpy(OUTLIN2.OUTLINA,"SE2ORDER=");
         strcpy(OUTLIN2.OUTLINB,GUIOA);
         SENDLIN2();
         SENDSTAT();
         TESTDIB();
    /*                                                       */
    /* ISSUE GNP REQUEST                                     */
    /*                                                       */
 do {
         strcpy(OUTLINE,GNP1);
         SENDLINE();
         EXEC DLI GET NEXT IN PARENT USING PCB(2)                  8
               INTO(&GNIOA) SEGLENGTH(60);
         strcpy(OUTLIN2.OUTLINA,"SEGMENT=");
         strcpy(OUTLIN2.OUTLINB,GNIOA);
         SENDLIN2();
         SENDSTAT();
         if (strncmp(dibptr->dibstat,"GE",2) != 0)               9
            TESTDIB();
  } while (strncmp(dibptr->dibstat,"GE",2) != 0);
    /*                                                       */
    /* ISSUE GN REQUEST                                      */
    /*                                                       */
         strcpy(OUTLINE,GN1);
```

```
     SENDLINE();
     EXEC DLI GET NEXT USING PCB(2)
           SEGMENT(SE2ORDER)                                          10
           INTO(&GNIOA) SEGLENGTH(60);
     strcpy(OUTLIN2.OUTLINA,"SE2ORDER=");
     strcpy(OUTLIN2.OUTLINB,GNIOA);
     SENDLIN2();
     SENDSTAT();
     TESTDIB();
/*                                                        */
/* INSERT SEGMENT                                         */
/*                                                        */
     strcpy(OUTLINE,INS1);
     SENDLINE();
     strcpy(NEWSEG.NEWSEGB,DATAB);                                  11
     strcpy(NEWSEG.NEWSEGC,DATAC);
     strcpy(ISRTIOA.ISRT1,NEWSEG.NEWSEGB);
     strcpy(ISRTIOA.ISRT2,NEWSEG.NEWSEGC);
     strcpy(OUTLIN3.OUTLINX,"ISRT SEG=");
     strcpy(OUTLIN3.OUTLINY,ISRTIOA.ISRT1);
     strcpy(OUTLIN3.OUTLINZ,ISRTIOA.ISRT2);
     SENDLIN3();
     EXEC DLI ISRT USING PCB(2)
           SEGMENT(SE2ORDER)
           FROM(&ISRTIOA) SEGLENGTH(60);
     SENDSTAT();
     if (strncmp(dibptr->dibstat,"II",2) == 0)
           strncpy(dibptr->dibstat,"  ",2);
     TESTDIB();
/*                                                        */
/* ISSUE GN REQUEST                                       */
/*                                                        */
     strcpy(OUTLINE,GN1);
     SENDLINE();
     EXEC DLI GET NEXT USING PCB(2)                                 12
           SEGMENT(SE2ORDER)
           INTO(&GNIOA) SEGLENGTH(60);
     strcpy(OUTLIN2.OUTLINA,"SE2ORDER=");
     strcpy(OUTLIN2.OUTLINB,GNIOA);
     SENDLIN2();
     SENDSTAT();
     TESTDIB();
/*                                                */
/* GET INSERTED SEGMENT TO BE REPLACED            */
/*                                                */
     strcpy(OUTLINE,GU2);
     SENDLINE();
     EXEC DLI GET UNIQUE USING PCB(2)                               13
           SEGMENT(SE2ORDER)
           WHERE(FE2OGREF="000999")
           INTO(&ISRTIOA) SEGLENGTH(60);
     strcpy(OUTLIN3.OUTLINX,"ISRT SEG=");
     strcpy(OUTLIN3.OUTLINY,ISRTIOA.ISRT1);
     strcpy(OUTLIN3.OUTLINZ,ISRTIOA.ISRT2);
     SENDLIN3();
     SENDSTAT();
     TESTDIB();
/*                                                        */
/* REPLACE SEGMENT                                        */
/*                                                        */
     strcpy(OUTLINE,REP1);
     SENDLINE();
     strcpy(REPLIOA.REPLIO1,DATAB);                                 14
     strcpy(REPLIOA.REPLIO2,"REGRUN REPLACED SEGMENT NO1.");
     strcpy(OUTLIN3.OUTLINX,"REPL SEG=");
     strcpy(OUTLIN3.OUTLINY,REPLIOA.REPLIO1);
     strcpy(OUTLIN3.OUTLINZ,REPLIOA.REPLIO2);
```

```
     SENDLIN3();
     EXEC DLI REPLACE USING PCB(2)
           SEGMENT(SE2ORDER)
           FROM(&REPLIOA) SEGLENGTH(60);
     SENDSTAT();
     TESTDIB();
/*                                                  */
/* ISSUE GN REQUEST                                 */
/*                                                  */
     strcpy(OUTLINE,GN1);
     SENDLINE();
     EXEC DLI GET NEXT USING PCB(2)                                  15
           SEGMENT(SE2ORDER)
           INTO(&GNIOA) SEGLENGTH(60);
     strcpy(OUTLIN2.OUTLINA,"SE2ORDER=");
     strcpy(OUTLIN2.OUTLINB,GNIOA);
     SENDLIN2();
     SENDSTAT();
     TESTDIB();
/*                                              */
/* GET REPLACED SEGMENT                         */
/*                                              */
     strcpy(OUTLINE,GU2);
     SENDLINE();
     EXEC DLI GET UNIQUE USING PCB(2)                                16
           SEGMENT(SE2ORDER)
           WHERE(FE2OGREF="000999")
           INTO(&REPLIOA) SEGLENGTH(60);
     strcpy(OUTLIN3.OUTLINX,"REPL SEG=");
     strcpy(OUTLIN3.OUTLINY,REPLIOA.REPLIO1);
     strcpy(OUTLIN3.OUTLINZ,REPLIOA.REPLIO2);
     SENDLIN3();
     SENDSTAT();
     TESTDIB();
/*                                                      */
/* ISSUE DELETE REQUEST                                 */
/*                                                      */
     strcpy(OUTLINE,DEL1);
     SENDLINE();
     strcpy(DLETIOA.DLET1,REPLIOA.REPLIO1);                          17
     strcpy(DLETIOA.DLET2,REPLIOA.REPLIO2);
     strcpy(OUTLIN3.OUTLINX,"DLET SEG=");
     strcpy(OUTLIN3.OUTLINY,DLETIOA.DLET1);
     strcpy(OUTLIN3.OUTLINZ,DLETIOA.DLET2);
     SENDLIN3();
     EXEC DLI DELETE USING PCB(2)
           SEGMENT(SE2ORDER)
           FROM(&DLETIOA) SEGLENGTH(60);
     SENDSTAT();
     TESTDIB();
/*                                                      */
/* ISSUE STAT REQUEST                                   */
/*                                                      */
     strcpy(OUTLINE,STAT);
     SENDLINE();
     EXEC DLI STAT USING PCB(2)                                      18
           VSAM FORMATTED
           INTO(&STATAREA);
     SENDSTT2();
     TESTDIB();
/*                                                      */
/* ISSUE TERM REQUEST                                   */
/*                                                      */
     strcpy(OUTLINE,TERM);
     SENDLINE();
     EXEC DLI TERM;                                                  19
     SENDSTAT();
```

```
         TESTDIB();
         strcpy(OUTLINE,DIVIDER);
         SENDLINE();
         SENDOK();
    /*                                                          */
    /* RETURN TO CICS                                           */
    /*                                                          */
         EXEC CICS RETURN;
}
    /*                                                          */
    /*                                                          */
    /*                                                          */
 SENDLINE()
 {
         EXEC CICS SEND FROM(OUTLINE) LENGTH(120);                          20
         EXEC CICS WRITEQ TD QUEUE("PRIM") FROM(OUTLINE) LENGTH(TLINE);
         strcpy(OUTLINE,BLANK);
         return;
 }

 SENDLIN2()
{
         EXEC CICS SEND FROM(OUTLIN2) LENGTH(120);
         EXEC CICS WRITEQ TD QUEUE("PRIM") FROM(OUTLIN2) LENGTH(TLINE);
         strcpy(OUTLIN2.OUTLINA,BLANK,9);
         strcpy(OUTLIN2.OUTLINB,BLANK,111);
         return;
 }

 SENDLIN3()
{
         EXEC CICS SEND FROM(OUTLIN3) LENGTH(120);
         EXEC CICS WRITEQ TD QUEUE("PRIM") FROM(OUTLIN3) LENGTH(TLINE);
         strcpy(OUTLIN3.OUTLINX,BLANK,9);
         strcpy(OUTLIN3.OUTLINY,BLANK,6);
         strcpy(OUTLIN3.OUTLINZ,BLANK,105);
         return;
 }

SENDSTAT()
{
         strncpy(OUTLIN2.OUTLINA,BLANK,9);
         strncpy(OUTLIN2.OUTLINB,BLAN2,110);
         strcpy(OUTLIN2.OUTLINA," DIBSTAT=");
         strcpy(OUTLIN2.OUTLINB,dibptr->dibstat);
         EXEC CICS SEND FROM(OUTLIN2) LENGTH(11);
         EXEC CICS WRITEQ TD QUEUE("PRIM") FROM(OUTLIN2) LENGTH(L11);
         strcpy(OUTLINE,DIVIDER);
         SENDLINE();
         return;
 }

SENDSTT2()
{
         strncpy(OUTLIN2.OUTLINA,BLANK,9);
         strncpy(OUTLIN2.OUTLINB,BLAN2,110);
         strcpy(OUTLIN2.OUTLINA," DIBSTAT=");
         strcpy(OUTLIN2.OUTLINB,dibptr->dibstat);
         EXEC CICS SEND FROM(STATAREA) LENGTH(360);
         EXEC CICS WRITEQ TD QUEUE("PRIM") FROM(STATAREA)
                   LENGTH(L360);
         return;
 }

SENDOK()
{
         EXEC CICS SEND FROM(OKMSG) LENGTH(120);
```

```
        EXEC CICS WRITEQ TD QUEUE("PRIM") FROM(OKMSG) LENGTH(TLINE);
        return;
 }

TESTDIB()                                                              21
{
    if (strncmp(dibptr->dibstat,"  ",2) == 0)
       return;
    else if (strncmp(dibptr->dibstat,"GK",2) == 0)
       return;
    else if (strncmp(dibptr->dibstat,"GB",2) == 0)
       return;
    else if (strncmp(dibptr->dibstat,"GE",2) == 0)
       return;
    else
       {
            EXEC CICS ABEND ABCODE("PETE");                            22
            EXEC CICS RETURN;
       }
         return;
 }
```

Notes for the sample C code:

1 You must include a standard header file `string.h` to gain access to string manipulation facilities.

2 You must include standard header file `stdio.h` to access the standard I/O library.

3 Define DL/I messages.

4 Define the I/O areas.

5 Program start.

6 Define PSB PC3COCHD.

7 Issue the first command. Retrieves the first occurrence of segment SE2ORDER and puts it into array OUTLIN2.

8 Issue the `GNP` command to get the next segment and put it into array OUTLIN2.

9 GE status codes indicate no more segments to get.

10 Get next segment SE2ORDER and put it into the array OUTLIN2.

11 Insert segment into array OUTLIN3.

12 Issue `GN` to retrieve next segment and put it into array OUTLIN2.

13 Get next segment that will be replaced and put it into OUTLIN3.

14 Replace the segment and put it into array OUTLIN3.

15 Get next segment and put it into array OUTLIN2.

16 Get the replaced segment and put it into array OUTLIN3.

17 Issue `DELETE` command after putting content of segment into array OUTLIN3.

18 Issue `STAT REQUEST` command.

19 Issue `TERM` command.

20 Output processing.

21 Check return code.

22 Do not code EXEC CICS commands in a batch or BMP program.

Preparing your EXEC DLI program for execution

You must translate, compile, and bind your EXEC DLI program before it can be executed.

You can use CICS-supplied procedures to translate, compile, and bind your program. The procedure you use depends on the type of program (batch, BMP, or CICS online) and the language it is written in (COBOL, PL/I, or assembler language).

The steps for preparing your program for execution are as follows:

1. Run the CICS command language translator to translate the EXEC DLI and EXEC CICS commands. COBOL, PL/I, and assembler language programs have separate translators.
2. Compile your program.
3. Bind:
   - An online program with the appropriate CICS interface module
   - A batch or BMP program with the IMS interface module.

Translator, compiler, and binder options required for EXEC DLI

To execute your EXEC DLI program, you must set the required translator, compile, and binder options.

Translator options required for EXEC DLI

Even when you use the CICS-supplied procedures for preparing your program, you must supply certain translator options.

For a CICS online program containing EXEC DLI commands, you must specify the DLI and CICS options. For a batch or BMP program containing EXEC DLI commands, you must specify the DLI option.

You can also specify the options on the EXEC job control statement that invokes the translator; if you use both methods, the CBL and *PROCESS statement overrides those in the EXEC statement. For more information on the translator options, see *CICS Transaction Server for z/OS CICS Application Programming Guide*.

You must ensure that the translator options you use in a COBOL program do not conflict with the COBOL compiler options.

Compiler options required for EXEC DLI

To compile your batch COBOL program, you may have to use different compiler options, depending on which COBOL compiler was chosen. For information on which compiler options should be used for a CICS program, see *CICS Transaction Server for z/OS CICS Application Programming Guide*.

Binder options required for EXEC DLI

If the compiler being used supports it, you can link a program written with EXEC commands as AMODE(31) RMODE(ANY).

Chapter 29. Defining application program elements

Define application program elements by using the EXEC DLI commands with the application interface block (AIB) and DL/I interface block (DIB), and by defining feedback and I/O areas.

Specifying an application interface block (AIB)

EXEC DLI commands can use the AIB interface.

For example, using the AIB interface, the format for the `GU` command would be `EXEC DLI GU AIB(aib)`, instead of `EXEC DLI GU USING PCB(n)` using the PCB format.

With IBM CICS Transaction Server for z/OS, the EXEC DLI commands are supported in the AIB format (as well as the PCB format). The AIB-only commands ICMD, RCMD, and GMSG are supported by using the EXEC DLI interface.

The CICS EDF (Execution Diagnostic Facility) debugging transaction supports AIB EXEC DLI requests, just as it handles PCB type requests.

AIB mask

The AIB mask must be supplied by the application and referenced in the EXEC call instead of the PCB number (for example, `EXEC DLI GU AIB(aib)`).

The DIBSTAT field is set with a valid STATUS code when AIBRETRN = X'00000000' or X'00000900'. Applications should test AIBRETRN for any other values and respond accordingly.

CICS restrictions with AIB support

Restrictions due to function shipping include:

- The AIBLEN field must be between 128 and 256 bytes. 128 bytes is recommended.
- LIST=NO must not be specified on any PCBs in the PSB.

Related reference:

Chapter 30, “EXEC DLI commands for an application program,” on page 509

Specifying the DL/I interface block (DIB)

Each time your program executes a DL/I command, DL/I returns a status code and other information to your program through the DL/I interface block (DIB), which is a subset of IMS PCB. Your program should check the status code to make sure the command executed successfully.

Each program's working storage contains its own DIB. The contents of the DIB reflect the status of the last DL/I command executed in that program. If the information in your program's DIB is required by another program used by your transaction, you must pass the information to that program.

To access fields in the DIB, use labels that are automatically generated in your program by the translator.

© Copyright IBM Corp. 1974, 2015

Restriction: These labels are reserved; you must not redefine them.

In your COBOL, PL/I, assembler language, and C programs, some variable names are mandatory.

For a COBOL program:

```
DIBVER    PICTURE X(2)
DIBSTAT    PICTURE X(2)
DIBSEGM    PICTURE X(8)
DIBSEGLV    PICTURE X(2)
DIBKFBL    PICTURE S9(4) COMPUTATIONAL
DIBDBDNM    PICTURE X(8)
DIBDBORG    PICTURE X(8)
```

```
DIBVER    CHAR(2)
DIBSTAT    CHAR(2)
DIBSEGM    CHAR(8)
DIBSEGLV    CHAR(2)
DIBKFBL    FIXED BINARY (15,0)
DIBDBDNM    CHAR(8)
DIBDBORG    CHAR(8)
```

For an assembler language program:

```
DIBVER    CL2
DIBSTAT    CL2
DIBSEGM    CL8
DIBSEGLV    CL2
DIBKFBL    H
DIBDBDNM    CL8
DIBDBORG    CL8
```

For a C program:

```
unsigned char     dibver    {2}  ;
unsigned char     dibstat   {2}  ;
unsigned char     dibsegm   {8}  ;
unsigned char     dibfic01  ;
unsigned char     dibfic02  ;
unsigned char     dibseglv  {2}  ;
signed short int  dibkfbl   ;
unsigned char     dibdbdnm  {8}  ;
unsigned char     dibdborg  {8}  ;
unsigned char     dibfic03  {6}  ;
```

The following notes explain the contents of each variable name. The name in parenthesis is the label used to access the contents.

1. **Translator Version (DIBVER)**

   This is the version of the DIB format your program is using. (DIBVER is used for documentation and problem determination.)

2. **Status Codes (DIBSTAT)**

   DL/I places a 2-character status code in this field after executing each DL/I command. This code describes the results of the command.

   After processing a DL/I command, DL/I returns control to your program at the next sequential instruction following the command. The first thing your program should do after each command is to test the status code field and take appropriate action. If the command was completely successful, this field contains blanks.

   The status codes that can be returned to this field (they are the only status codes returned to your program) are:

   bb (Blanks) The command was completely successful.

BA For GU, GN, GNP, DLET, REPL, and ISRT commands. Data was unavailable.

BC For DLET, REPL, and ISRT commands. A deadlock was detected.

FH For GU, GN, GNP, DLET, REPL, ISRT, POS, CHKP, and SYMCHKP commands. The DEDB was inaccessible.

FW For GU, GN, GNP, DLET, REPL, ISRT, and POS commands. More buffer space is required than normally allowed.

GA For unqualified GN and GNP commands. DL/I returned a segment, but the segment is at a higher level in the hierarchy than the last segment that was returned.

GB For GN commands. DL/I reached the end of the database trying to satisfy your GN command and did not return a segment to your program's I/O area.

GD For ISRT commands. The program issued an ISRT command that did not have SEGMENT options for all levels above that of the segment being inserted.

GE For GU, GN, GNP, ISRT, and STAT commands. DL/I was unable to find the segment you requested, or one or more of the parents of the segment you are trying to insert.

GG For Get commands. DL/I returns a GG status code to a program with a processing option of GOT or GON when the segment that the program is trying to retrieve contains an invalid pointer.

GK For unqualified GN and GNP commands. DL/I returned a segment that satisfies an unqualified GN or GNP request, but the segment is of a different segment type (but at the same level) than the last segment returned.

II For ISRT commands. The segment you are trying to insert already exists in the database. This code can also be returned if you have not established a path for the segment before trying to insert it. The segment you are trying to insert might match a segment with the same key in another hierarchy or database record.

LB For load programs only after issuing a LOAD command. The segment you are trying to load already exists in the database. DL/I returns this status code only for segments with key fields.

NI For ISRT and REPL commands. The segment you are trying to insert or replace requires a duplicate entry to be inserted in a secondary index that does not allow duplicate entries. This status code is returned for batch programs that write log records to direct access storage. If a CICS program that does not log to disk encounters this condition, the program (transaction) is abnormally terminated.

TG For TERM commands. The program tried to terminate a PSB when one was not scheduled.

The listed status codes (DIBSTAT) indicate exceptional conditions, and are the only status codes returned to your program. All other status codes indicate error conditions and cause your transaction or batch program to abnormally terminate. If you want to pass control to an error routine from your CICS program, you can use the CICS HANDLE ABEND command; the last 2 bytes of the abend code are the IMS status code that caused the abnormal termination. For

more information on the HANDLE ABEND command, see the application programming reference manual for your version of CICS. Batch BMP programs abend with abend 1041.

3. **Segment Name (DIBSEGM)**

   This is the name of the lowest-level segment successfully accessed. When a retrieval is successful, this field contains the name of the retrieved segment. If the retrieval is unsuccessful, this field contains the last segment, along the path to the requested segment, that satisfies the command.

   After issuing an XRST command, this field is either set to blanks (indicating a successful normal start), or a checkpoint ID (indicating the checkpoint ID from which the program was restarted).

   You should test this field after issuing any of the following commands:

   - GN
   - GNP
   - GU
   - ISRT
   - LOAD
   - RETRIEVE
   - XRST

4. **Segment Level Number (DIBSEGLV)**

   This is the hierarchic level of the lowest-level segment retrieved. When IMS DB retrieves the segment you have requested, IMS DB places, in character format, the level number of that segment in this field. If you are issuing a path command, IMS DB places the number of the lowest-level segment retrieved. If IMS DB is unable to find the segment you have requested, it gives the level number of the last segment it encountered that satisfied your command. This is the lowest segment on the last path that IMS DB encountered while searching for the segment you requested.

   You should test this field after issuing any of the listed commands:

   - GN
   - GNP
   - GU
   - ISRT
   - LOAD
   - RETRIEVE

5. **Key Feedback Length (DIBKFBL)**

   This is a halfword field that contains the length of the concatenated key when you use the KEYFEEDBACK option with get commands. If your key feedback area is not long enough to contain the concatenated key, the key is truncated, and this area indicates the actual length of the full concatenated key.

6. **Database Description Name (DIBDBDNM)**

   This is the fullword field that contains the name of the DBD. The DBD is the DL/I control block that contains all information used to describe a database. The DIBDBDNM field is returned only on a QUERY command.

7. **Database Organization (DIBDBORG)**

   This is the fullword field that names the type of database organization (HDAM, HIDAM, HISAM, HSAM, GSAM, SHSAM, INDEX, or DEDB) padded to the right with blanks. The DIBDBORG field is returned only on a QUERY command.

Defining a key feedback area

To retrieve the concatenated key of a segment, you must define an area into which the key is placed.

The concatenated key returned is that of the lowest-level segment retrieved. (The segment retrieved is indicated in the DIB by the DIBSEGM and DIBSEGLV fields.)

Specify the name of the area using the KEYFEEDBACK option on a GET command.

A concatenated key is made up of the key of a segment, plus the keys for all of its parents. For example, say you requested the concatenated key of the ILLNESS segment for January 2, 1988, for patient number 05142. **0514219880102** would be returned to your key feedback field. This number includes the key field of the ILLNESS segment, ILLDATE, concatenated to the key field of the PATIENT segment, PATNO.

If you define an area that is not long enough to contain the entire concatenated key, the key is truncated.

Defining I/O areas

Use I/O areas to pass segments back and forth between your program and the database.

The contents of an I/O area depends on the kind of command you are issuing:

- When you retrieve a segment, DL/I places the segment you requested in the I/O area.
- When you add a new segment, you build the new segment in the I/O area before issuing an `ISRT` command.
- Before you modify a segment, you first retrieve the segment into the I/O area then issue the `DLET` or `REPL` command.

Restriction: The I/O area must be long enough to contain the longest segment you retrieve from or add to the database. (Otherwise, you might experience storage overlap.) If you are retrieving, adding, or replacing multiple segments in one command, you must define an I/O area for each segment.

As an example of what a segment looks like in your I/O area, say that you retrieved the ILLNESS segment for Robert James, who came to the clinic on March 3, 1988. He was treated for strep throat. The data returned to your I/O area would look like this:

```
19880303STREPTHROA
```

COBOL I/O area

The I/O area in a COBOL program should be defined as a 01 level working storage entry. You can further define the area with 02 entries.

```
IDENTIFICATION DIVISION.
⋮
DATA DIVISION.
WORKING-STORAGE SECTION.
01    INPUT-AREA.
    02 KEY PICTURE X(6).
    02 FIELD PICTURE X(84).
```

PL/I I/O area

In PL/I, the name for the I/O area used in the DL/I call can be the name of a fixed-length character string, a major structure, a connected array, or an adjustable character string.

Restriction: The PL/I I/O area *cannot* be the name of a minor structure or a character string with the attribute VARYING. If you want to define it as a minor structure, you can use a pointer to the minor structure as the parameter.

Your program should define the I/O area as a fixed-length character string and pass the name of that string, or define it in one of the other ways described previously and then pass the pointer variable that points to that definition. If you want to use substructures or elements of an array, use the DEFINED or BASED attribute.

```
DECLARE    1 INPUT_AREA,
           2 KEY CHAR(6),
           2 FIELD CHAR(84);
```

Assembler language I/O area

The I/O area in an assembler language program is formatted as follows:

```
IOAREA   DS   0CL90
KEY      DS   CL6
FIELD    DS   CL84
```

Chapter 30. EXEC DLI commands for an application program

The EXEC DLI commands in your application program is used together with a program specification block (PSB) and different kinds of program communication blocks (PCBs).

Related reference:

"Specifying an application interface block (AIB)" on page 503

PCBs and PSB

A program specification block (PSB) used in a DBCTL environment can contain I/O PCBs, alternate PCBs, database PCBs (DB PCB), or GSAM PCBs.

I/O PCB

In a DBCTL environment, an I/O PCB is needed to issue DBCTL service requests. Unlike the other types of PCB, it is not defined with PSB generation, but if the application program is using an I/O PCB, this has to be indicated in the PSB scheduling request.

Alternate PCB

An alternate PCB defines a logical terminal and can be used instead of the I/O PCB when it is necessary to direct a response to a terminal. Alternate PCBs appear in PSBs used in a CICS-DBCTL environment, but are used only in an IMS DC environment. CICS applications using DBCTL cannot successfully issue commands that specify an alternate PCB, an MSDB PCB, or a GSAM PCB. However, a PSB that contains PCBs of these types can be scheduled successfully in a CICS-DBCTL environment.

Alternate PCBs are included in the PCB address list returned to a call level application program. In an EXEC DLI application program, the existence of alternate PCBs in the PSB affects the PCB number used in the PCB keyword.

DB PCB

A DB PCB is the PCB that defines an application program's interface to a database. One DB PCB is needed for each database view used by the application program. It can be a full-function PCB, a DEDB PCB, or an MSDB PCB.

GSAM PCB

A GSAM PCB defines an application program's interface for GSAM operations.

When using DBCTL, a CICS program receives, by default, a DB PCB as the first PCB in the parameter list passed to it after scheduling. However, when your application program can handle an I/O PCB, you indicate this using the SYSSERVE keyword on the `SCHD` command. The I/O PCB is then the first PCB in the parameter address list passed back to your application program.

© Copyright IBM Corp. 1974, 2015

I/O PCBs and alternate PCBs in various types of application programs

DB batch programs

Alternate PCBs are always included in the list of PCBs supplied to the program by DL/I irrespective of whether you have specified CMPAT=Y. The I/O PCB is returned depending on the CMPAT option.

If you specify CMPAT=Y, the PCB list contains the address of the I/O PCB, followed by the addresses of any alternate PCBs, followed by the addresses of any DB PCBs.

If you do not specify CMPAT=Y, the PCB list contains the addresses of any alternate PCBs followed by the addresses of the DB PCBs.

BMP programs, MPPs, and IFPs

I/O PCBs and alternate PCBs are always passed to BMP programs. I/O PCBs and alternate PCBs are also always passed to MPPs and to IFP application programs.

The PCB list contains the address of the I/O PCB, followed by the addresses of any alternate PCBs, followed by the addresses of the DB PCBs.

CICS programs with DBCTL

The first PCB always refers to the first DB PCB whether you specify the SYSSERVE keyword.

The following table summarizes the I/O PCB and alternate PCB information. The first column lists different DB environments, the second and third column specify if the I/O PCB or alternate PCB, respectively, is valid in the specified environment.

Table 87. Summary of PCB information

Environment	EXEC DLI: I/O PCB count included in PCB(n)	EXEC DLI: Alternate PCB count included in PCB(n)
CICS DBCTL[1]	No	No
CICS DBCTL[2]	No	No
BMP	Yes	Yes
Batch[3]	No	Yes
Batch[4]	Yes	Yes

Notes:

1. SCHD command issued without the SYSSERVE option.
2. SCHD command issued with the SYSSERVE option for a CICS DBCTL command or for a function-shipped command which is satisfied by a remote CICS system using DBCTL.
3. CMPAT=N specified on the PSBGEN statement.
4. CMPAT=Y specified on the PSBGEN statement.

Format of a PSB

The following is the format of a PSB.

```
[IOPCB]
[Alternate PCB ... Alternate PCB]
[DBPCB ... DBPCB]
[GSAMPCB ... GSAMPCB]
```

Each PSB must contain at least one PCB. The I/O PCB must be addressable in order to issue a system service command. An alternate PCB is used only for IMS online programs, which can run only with the Transaction Manager. Alternate PCBs can be present even though your program does not run under the Transaction Manager. A DB PCB can be a full-function PCB, a DEDB PCB, or an MSDB PCB.

Chapter 31. Recovering databases and maintaining database integrity

You can issue these commands to recover data accessed by your program and maintain data integrity.

- The Basic Checkpoint command, CHKP, which you can use to issue checkpoints from a batch or BMP program
- The Symbolic Checkpoint command, SYMCHKP, which you can use to issue checkpoints from a batch or BMP program and to specify data areas that can be restored when you restart your program
- The Extended Restart command, XRST, which you can use along with symbolic checkpoints to start or restart your batch or BMP program
- The rollback commands, ROLL and ROLB, which you can use to dynamically back out database changes from a batch or BMP program
- The managing-backout-points commands, SETS and ROLS, which you can use to set multiple backout points and then return to these points later
- The Dequeue command, DEQ, which releases previously reserved segments

To use any of the commands, you must have defined an I/O PCB for your program, except for the DEDB DEQ calls, which are issued against a DEDB PCB.

Issuing checkpoints in a batch or BMP program

The two kinds of commands that allow you to make checkpoints are: the CHKP, or Basic Checkpoint command, and the SYMCHKP, or Symbolic Checkpoint command.

Batch programs can use either the Symbolic Checkpoint or the Basic Checkpoint command.

Both checkpoint commands make it possible for you to commit your program's changes to the database and to establish places from which the batch or BMP program can be restarted, in cases of abnormal termination.

Requirement: You must not use the CHKPT=EOV parameter on any DD statement to take an IMS checkpoint.

Because both checkpoint commands cause a loss of database position at the time the command is issued, you must reestablish position with a GU command or other methods.

You cannot reestablish position in the midst of nonunique keys or nonkeyed segments.

Issuing the CHKP command

When you issue a CHKP command, you must provide the code for restarting your program and you must specify the ID for the checkpoint. You can supply either the name of a data area in your program that contains the ID, or you can supply the actual ID, enclosed in single quotation marks. For example, either of the following commands is valid:

```
EXEC DLI CHKP ID(chkpid);
```

© Copyright IBM Corp. 1974, 2015

```
EXEC DLI CHKP ID('CHKP0007');
```

Issuing the SYMCHKP command

The SYMCHKP command in batch and BMP programs:

- Works with the Extended Restart (XRST) command to restart your program if it terminates abnormally.
- Can save as many as seven program data areas, which are restored when your program is restarted. You can save variables, counters, and status information.

For examples of how to specify the SYMCHKP command, see the topic "SYMCHKP Command" in *IMS Version 11 Application Programming APIs*.

Restarting your program and checking for position

Programs that issue Symbolic Checkpoint commands must also issue the Extended Restart (XRST) command. You must issue XRST once, as the first command in the program. You can use the XRST command to start your program normally, or to restart it in case of an abnormal termination.

You can restart your program from one of the following:

- A specific checkpoint ID
- A time/date stamp

Because the XRST command attempts to reposition the database, your program also needs to check for correct position.

Backing out database updates dynamically: the ROLL and ROLB commands

When a batch program determines that some of its processing is invalid, the ROLL and ROLB commands make it possible for the program to remove the effects of its inaccurate processing.

You can use both ROLL and ROLB in batch programs. You can only use the ROLB command in batch programs if the system log is stored on direct access storage and if you have specified BKO=Y in the parm field of your JCL.

Issuing either of these commands causes DL/I to back out any changes your program has made to the database since its last checkpoint, or since the beginning of the program if your program has not issued a checkpoint.

Using intermediate backout points: the SETS and ROLS commands

Use the SETS and ROLS commands to define multiple points at which to preserve the state of DL/I full-function databases and to return to these points later. For example, you can use them to allow your program to handle situations that can occur when PSB scheduling complete without all of the referenced DL/I databases being available.

The SETS and ROLS commands apply only to DL/I full-function databases. Therefore, if a logical unit of work (LUW) is updating recoverable resources other than full-function databases (VSAM files, for example), the SETS and ROLS requests have no effect on the non-DL/I resources. The backout points are *not* CICS commit

points; they are intermediate backout points that apply only to DBCTL resources. Your program must ensure the consistency of all the resources involved.

Before initiating a set of DL/I requests to perform a function, you can use a SETS command to define points in your application at which to preserve the state of DL/I databases. Your application can issue a ROLS command later if it cannot complete the function. You can use the ROLS command to back out to the state all full-function databases were in before either a specific SETS request or the most recent commit point.

Chapter 32. Processing Fast Path databases

Using EXEC DLI commands under DBCTL, a CICS program or a batch-oriented BMP program can access DEDBs. Parameters allow your program to use facilities of the DEDBs such as subset pointers.

A DEDB contains a root segment and as many as 127 types of dependent segment. One of these types can be a sequential dependent; the other 126 are direct dependents. Sequential dependent segments are stored in chronological order. Direct dependent segments are stored hierarchically.

DEDBs provide high data availability. Each DEDB can be partitioned, or divided into multiple *areas*. Each area contains a different set of database records. In addition, you can make up to seven copies of each area data set. If an error exists in one copy of an area, application programs can access the data by using another copy of that area. This is transparent to the application program. When an error occurs to data in a DEDB, IMS does not stop the database. It makes the data in error unavailable, but continues to schedule and process application programs. Programs that do not need the data in error are unaffected.

DEDBs can be shared among application programs in separate IMS systems. Sharing DEDBs is virtually the same as sharing full-function databases, and most of the same rules apply. IMS systems can share DEDBs at the area level (instead of at the database level as with full-function databases), or at the block level.

Processing Fast Path DEDBs with subset pointer options

Subset pointers and the options you use with them are optimization tools that significantly improve the efficiency of your program when you need to process long segment chains.

Subset pointers divide a chain of segment occurrences under the same parent into two or more groups, or subsets. You can define as many as eight subset pointers for any segment type. You then define the subset pointers from within an application program. Each subset pointer points to the start of a new subset. For example, in the following figure, suppose you defined one subset pointer that divided the last three segment occurrences from the first four. Your program can then refer to that subset pointer through options, and directly retrieve the last three segment occurrences.

© Copyright IBM Corp. 1974, 2015

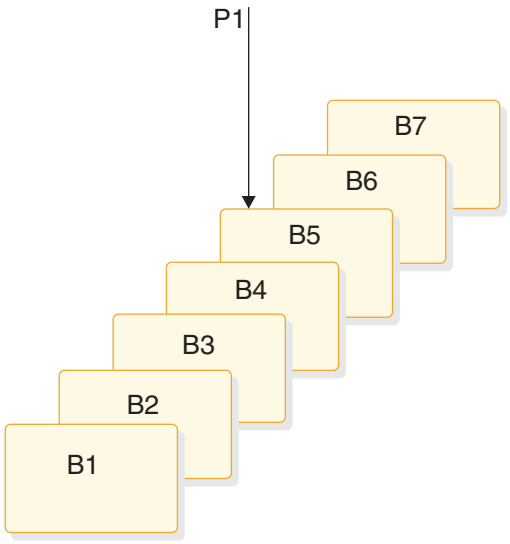


Figure 88. Processing a long chain of segment occurrences with subset pointers

You can use subset pointers at any level of the database hierarchy, except at the root level. Subset pointers used for the root level are ignored.

The next two figures show some of the ways you can set subset pointers. Subset pointers are independent of one another, which means that you can set one or more pointers to any segment in the chain. For example, you can set more than one subset pointer to a segment, as shown in the following figure.

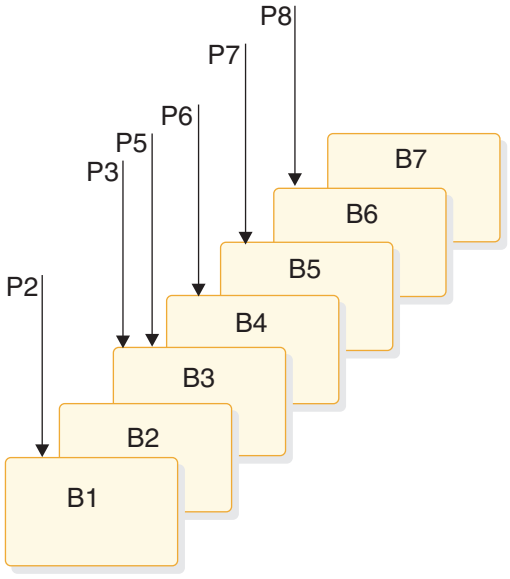


Figure 89. Examples of setting multiple subset pointers

Alternatively, you can define a one-to-one relationship between the pointers and the segments, as shown in Figure 90 on page 519 where each segment occurrence has one subset pointer.

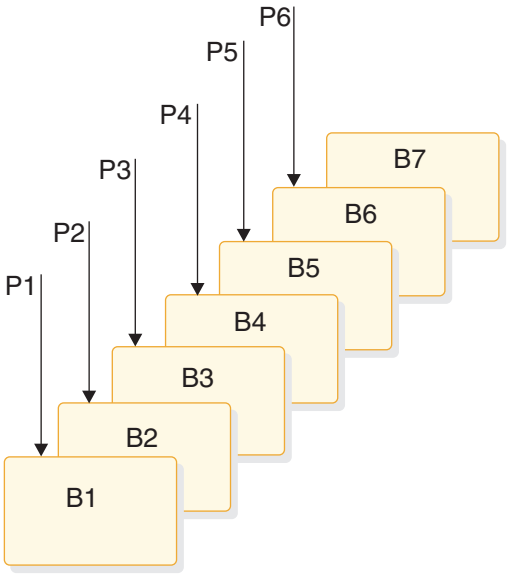


Figure 90. More examples of setting subset pointers

The following figure shows how the use of subset pointers divides a chain of segment occurrences under the same parent into subsets. Each subset ends with the last segment in the entire chain. For example, the last segment in the subset defined by subset pointer 1 is B7.

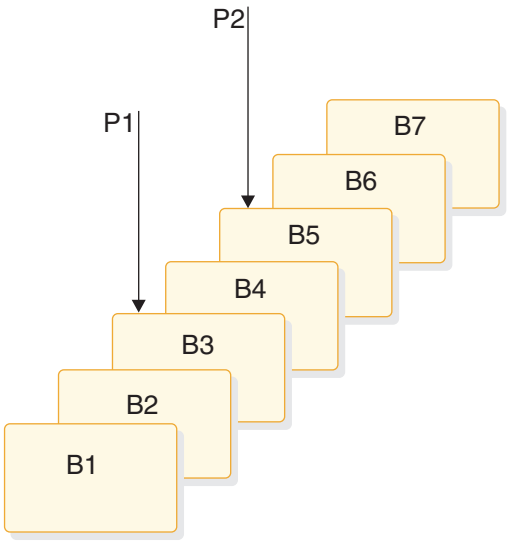


Figure 91. How subset pointers divide a chain into subsets

Preparing to use subset pointers

For your program to use subset pointers, the pointers must be defined in the DBD for the DEDB, and in your program's PSB.

In the DBD, you specify the number of pointers for a segment chain. You can specify as many as eight pointers for any segment chain.

In the PSB, you specify which pointers your program uses; you define this on the SENSEG statement. (Each pointer is defined as an integer from 1 to 8.) You also specify on the SENSEG statement whether your program can set the pointers it

uses. If your program has read-only sensitivity, it cannot set pointers, but can only retrieve segments using subset pointers already set. If your program has update sensitivity, it can update subset pointers by using the SET, SETCOND, MOVENEXT, and SETZERO options.

After the pointers are defined in the DBD and the PSB, an application program can set the pointers to segments in a chain. When an application program finishes executing, the subset pointers used by that program remain as they were set by the program and are not reset.

Designating subset pointers

To use subset pointers in your program, you must know the numbers for the pointers as they were defined in the PSB.

Then, when you use the subset pointer options, you specify the number for each subset pointer you want to use immediately after the option; for example, you would use P3 to indicate that you want to retrieve the first segment occurrence in the subset defined by subset pointer 3. No default exists, so if you do not include a number between 1 and 8, IMS considers your qualification statement invalid and returns an AJ status code to your program.

Subset pointer options

To take advantage of subsets, application programs use five different options.

The options are:

GETFIRST
Allows you to retrieve the first segment in a subset.

SETZERO
Sets a subset pointer to zero.

MOVENEXT
Sets a subset pointer to the segment following the current segment. Current position is at the current segment.

SET Unconditionally sets a subset pointer to the current segment. Current position is at the current segment.

SETCOND
Conditionally sets a subset pointer to the current segment. Current position is at the current segment.

Banking transaction application example

The examples in this chapter are based on a sample application, the recording of banking transactions for a passbook account. The transactions are written to a database as either posted or unposted, depending on whether they were posted to the customer's passbook. For example, when Bob Emery does business with the bank, but forgets to bring in his passbook, an application program writes the transactions to the database as unposted. The application program sets a subset pointer to the first unposted transaction, so it can be easily accessed later. The next time Bob remembers to bring in his passbook, a program posts the transactions. The program can directly retrieve the first unposted transaction using the subset pointer that was previously set. After the program has posted the transactions, it sets the subset pointer to zero; an application program that subsequently updates the database can determine that no unposted transactions exist. The following

figure summarizes the processing performed when the passbook is unavailable.

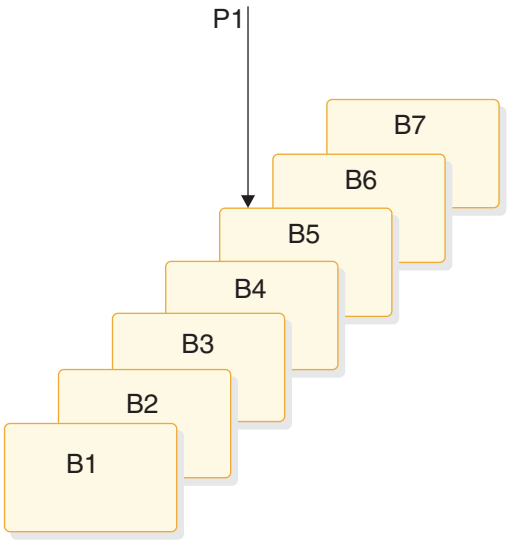


Figure 92. Processing performed for the sample passbook example when the passbook is unavailable

When the passbook is available, an application program adds the unposted transactions to the database, setting subset pointer 1 to the first unposted transaction. The following figure summarizes the processing performed when the passbook is available.

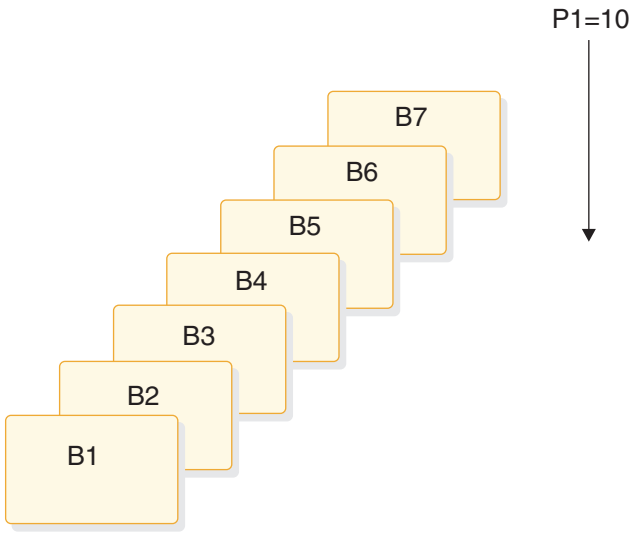


Figure 93. Processing performed for the sample passbook example when the passbook is available

When the passbook is available, an application program retrieves the first unposted transaction using the program, then posts all unposted transactions, setting subset pointer 1 to zero.

GETFIRST option: retrieving the first segment of a subset

To retrieve the first segment occurrence in the subset, your program issues a Get command with the GETFIRST option. The GETFIRST option does not set or move

the pointer, but indicates to IMS that you want to establish position on the first segment occurrence in the subset. The GETFIRST option is like the FIRST option, except that the GETFIRST option applies to the subset instead of to the entire segment chain.

Using the previous example, imagine that Bob Emery visits the bank with his passbook and you want to post all of the unposted transactions. Because subset pointer 1 was previously set to the first unposted transaction, your program can use the following command to retrieve that transaction:

```
EXEC DLI GU SEGMENT(A) WHERE(AKEY = 'A1')
     SEGMENT(B) INTO(BAREA) GETFIRST('1');
```

As shown in following figure, this command retrieves segment B5. To continue processing segments in the chain, you can issue Get Next commands, as you would if you were not using subset pointers.

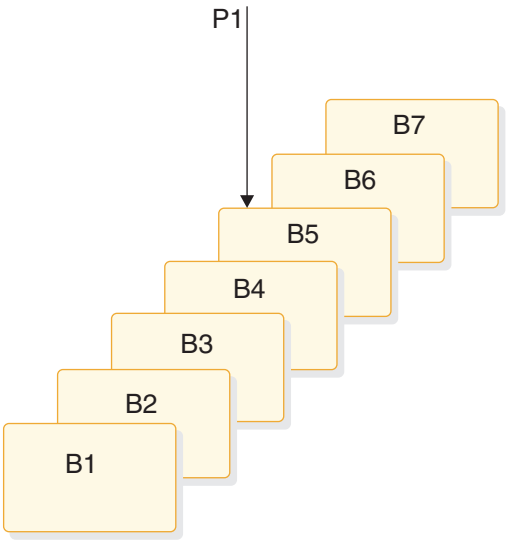


Figure 94. Retrieving the first segment in a chain of segments

If the subset does not exist (subset pointer 1 has been set to zero), IMS returns a GE status code, and your position in the database immediately follows the last segment in the chain. Using the passbook example, the GE status code indicates that no unposted transactions exist.

You can specify only one GETFIRST option per qualification statement; if you use more than one GETFIRST in a qualification statement, IMS returns an AJ status code to your program. The rules for using the GETFIRST option are:

1. You can use GETFIRST with all options except:
   - FIRST
   - LOCKCLASS
   - LOCKED
2. Other options take effect after the GETFIRST option has, and position has been established on the first segment in the subset.
3. If you use GETFIRST with LAST, the last segment in the segment chain is retrieved.
4. If the subset pointer specified with GETFIRST is not set, IMS returns a GE status code, not the last segment in the segment chain.

5. Do not use GETFIRST with FIRST. This causes you to receive an AJ status code.
6. GETFIRST overrides all insert rules, including LAST.

SETZERO, MOVENEXT, SET, and SETCOND options: setting the subset pointers

The SETZERO, MOVENEXT, SET, and SETCOND options allow you to redefine subsets by modifying the subset pointers. Before your program can set a subset pointer, it must establish a position in the database. A command must be fully satisfied before a subset pointer is set. The segment a pointer is set to depends on your current position at the completion of the command. If a command to retrieve a segment is not completely satisfied, and a position is not established, the subset pointers remain as they were before the command was issued.

- **Setting the subset pointer to zero: SETZERO**

  The SETZERO option sets the value of the subset pointer to zero. After your program issues a command with the SETZERO option, the pointer is no longer set to a segment; the subset defined by that pointer no longer exists. (IMS returns a status code of GE to your program if you try to use a subset pointer having a value of zero.)

  Using the previous example, say that you used the GETFIRST option to retrieve the first unposted transaction. You would then process the chain of segments, posting the transactions. After posting the transactions and inserting any new ones into the chain, you would use the SETZERO option to set the subset pointer to zero as shown in the following command:

  ```
  EXEC DLI ISRT SEGMENT(A) WHERE(AKEY = 'A1')
       SEGMENT(B) FROM(BAREA) SETZERO('1');
  ```

  After this command, subset pointer 1 would be set to zero, indicating to a program updating the database later on that no unposted transactions exist.

- **Moving the subset pointer forward to the next segment after your current position: MOVENEXT**

  To move the subset pointer forward to the next segment after your current position, your program issues a command with the MOVENEXT option. Using the previous example, say that you wanted to post some of the transactions, but not all, and that you wanted the subset pointer to be set to the first unposted transaction. The following command sets subset pointer 1 to segment B6.

  ```
  EXEC DLI GU SEGMENT(A) WHERE(AKEY = 'A1')
       SEGMENT(B) INTO(BAREA) GETFIRST('1') MOVENEXT('1');
  ```

  The process of moving the subset pointer with this command is shown in the following figure. If the current segment is the last in the chain, and you use a MOVENEXT option, IMS sets the pointer to zero.

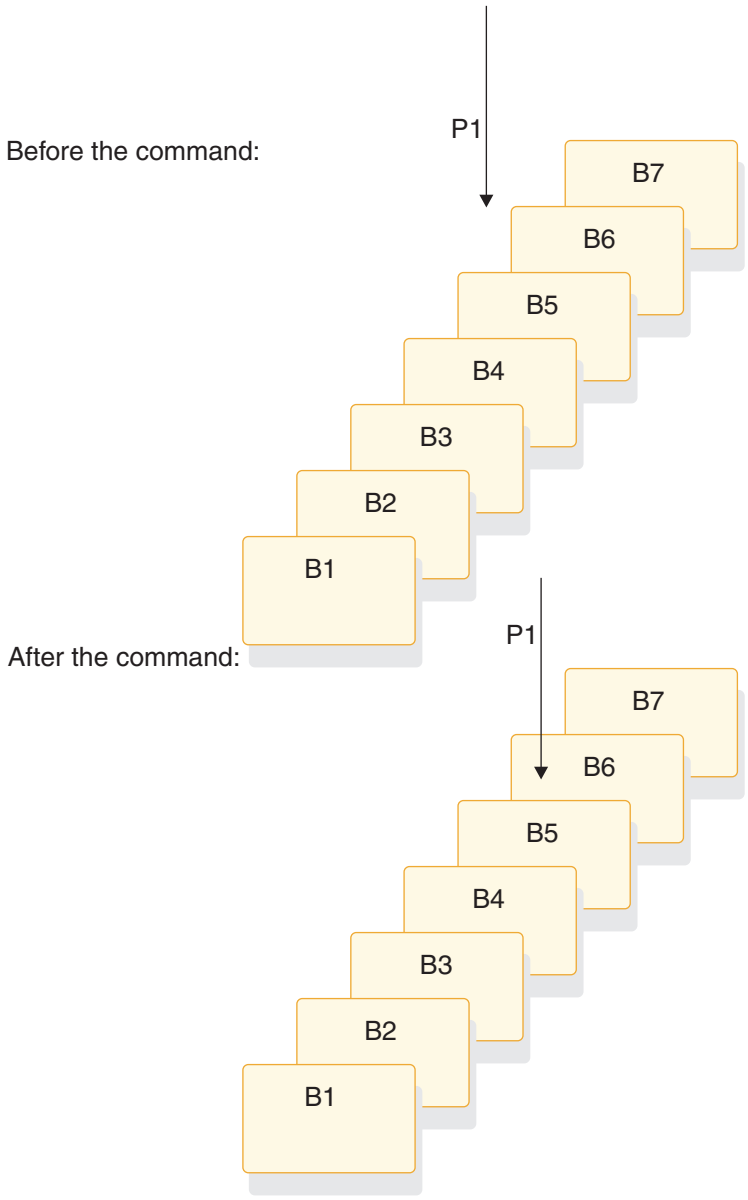


Figure 95. Moving the subset pointer to the next segment after your current position

- **Setting the subset pointer unconditionally: SET**

  You use the SET option to set a subset pointer. The SET option sets a subset pointer unconditionally, regardless of whether or not it is already set. When your program issues a command that includes the SET option, IMS sets the pointer to your current position.

  For example, to retrieve the first B segment occurrence in the subset defined by subset pointer 1, and to reset pointer 1 at the next B segment occurrence, you would issue the following commands:

```
EXEC DLI GU SEGMENT(A) WHERE(AKEY = 'A1')
     SEGMENT(B) INTO(BAREA) GETFIRST('1');
EXEC DLI GN SEGMENT(B) INTO(BAREA) SET('1');
```

  After you have issued these commands, instead of pointing to segment B5, subset pointer 1 points to segment B6, as shown in the following figure.

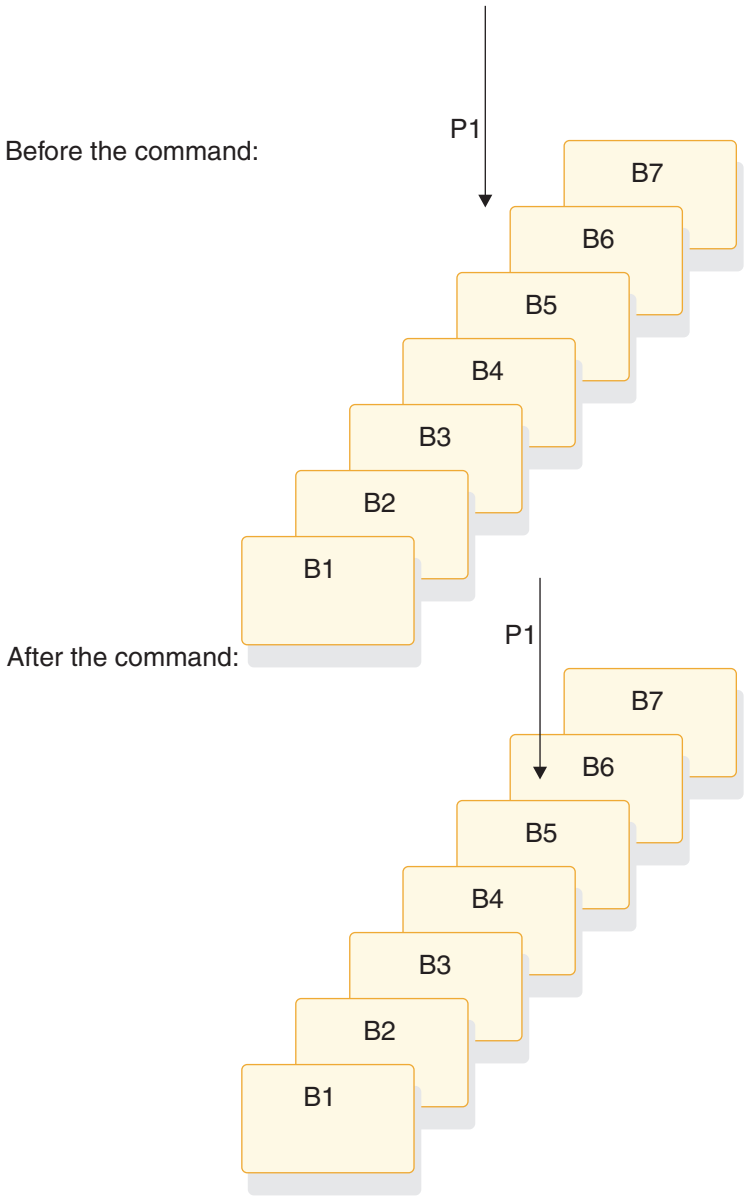


Figure 96. Unconditionally setting the subset pointer to your current position

- **Setting the subset pointer conditionally: SETCOND**

  Your program uses the SETCOND option to conditionally set the subset pointer. The SETCOND option is similar to the SET option; the only difference is that, with the SETCOND option, IMS updates the subset pointer only if the subset pointer is *not* already set to a segment.

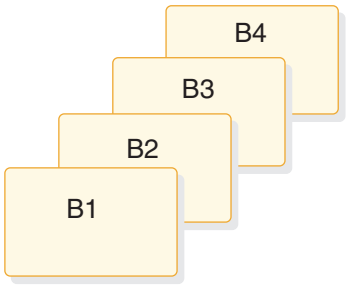
  Using the passbook example, say that Bob Emery visits the bank and forgets to bring his passbook; you add the unposted transactions to the database. You want to set the pointer to the first unposted transaction so that when you post the transactions later, you can immediately access the first one. The following command sets the subset pointer to the transaction you are inserting, if it is the first unposted one:

```
EXEC DLI ISRT SEGMENT(A) WHERE(AKEY = 'A1')
     SEGMENT(B) FROM(BAREA) SETCOND('1');
```

  As shown by the following figure, this command sets subset pointer 1 to segment B5. If unposted transactions already existed, the subset pointer is not

changed.

Before the command:



After the command:

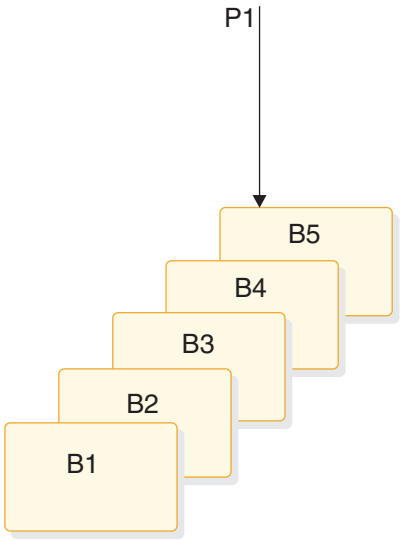


Figure 97. Conditionally setting the subset pointer to your current position

Inserting segments in a subset

When you use the GETFIRST option to insert an unkeyed segment in a subset, the new segment is inserted before the first segment occurrence in the subset. However, the subset pointer is not automatically set to the new segment occurrence. For example, the following command inserts a new B segment occurrence in front of segment B5, but does not set subset pointer 1 to point to the new B segment occurrence:

```
EXEC DLI ISRT SEGMENT(A) WHERE(AKEY = 'A1')
     SEGMENT(B) FROM(BAREA) GETFIRST('1');
```

To set subset pointer 1 to the new segment, you use the SET option along with the GETFIRST option, as shown in the following example:

```
EXEC DLI ISRT SEGMENT(A) WHERE(AKEY = 'A1')
     SEGMENT(B) FROM(BAREA) GETFIRST('1') SET ('1');
```

If the subset does not exist (subset pointer 1 has been set to zero), the segment is added to the end of the segment chain.

Deleting the segment pointed to by a subset pointer

If you delete the segment pointed to by a subset pointer, the subset pointer points to the next segment occurrence in the chain. If the segment you delete is the last in the chain, the subset pointer is set to zero.

Combining options

You can use the SET, MOVENEXT, and SETCOND options with other options, and you can combine subset pointer options with each other, provided they do not conflict. For example, you can use GETFIRST and SET together, but you cannot use SET and SETZERO together because their functions conflict. If you combine options that conflict, IMS returns an AJ status code to your program.

You can use one GETFIRST option per qualification statement, and one update option (SETZERO, MOVENEXT, SET, or SETCOND) for each subset pointer.

Subset pointer status codes

If you make an error in a qualification statement that contains subset pointer options, IMS can return these status codes to your program.

AJ The qualification statement used a GETFIRST, SET, SETZERO, SETCOND, or MOVENEXT option for a segment for which there are no subset pointers defined in the DBD.

The subset options included in the qualification statement are in conflict; for example, if one qualification statement contained a SET option and a SETZERO option for the same subset pointer, IMS would return an AJ status code. S means to set the pointer to current position; Z means to set the pointer to zero. You cannot use these options together in one qualification statement.

The qualification statement included more than one GETFIRST option.

The pointer number following a subset pointer option is invalid. You either did not include a number, or included an invalid character. The number following the option must be between 1 and 8, inclusive.

AM The subset pointer referenced in the qualification statement was not specified in the program's PSB. For example, if your program's PSB specifies that your program can use subset pointers 1 and 4, and your qualification statement referenced subset pointer 5, IMS would return an AM status code to your program.

Your program tried to use an option that updates the pointer (SET, SETCOND, or MOVENEXT) but the program's PSB did not specify pointer update sensitivity.

Your program attempted to open a GSAM database without specifying an IOAREA.

The POS command

You can use the Position (POS) command (only with DEDBs) to perform the following functions.

- Retrieve the location of a specific sequential dependent segment, or retrieves the location of the last inserted sequential dependent segment.
- Tell you the amount of unused space within each DEDB area. For example, you can use the position information that IMS returns for a POS command to scan or delete the sequential dependent segments for a particular time period.

For the syntax of the POS command, see the topic "POS Command" in *IMS Version 11 Application Programming APIs*.

If the area the POS command specifies is unavailable, the I/O area is unchanged and the status code FH is returned.

Related reference:

POS command (Application Programming APIs)

Locating a specific sequential dependent segment

When you have position on a particular root segment, you can retrieve the position information and the area name of a specific sequential dependent segment of that root.

If you have a position established on a sequential dependent segment, the search starts from that position. IMS returns the position information for the first sequential dependent segment that satisfies the command.

To retrieve this information, you issue a POS command with a qualification statement containing the segment name of the sequential dependent. The current position after this kind of POS command is in the same place that it is after a GNP command.

After a successful POS command, the I/O area contains:

LL A 2-byte field giving the total length of the data in the I/O area, in binary.

Area Name
An 8-byte field giving the ddname from the AREA statement.

Position
An 8-byte field containing the position information for the requested segment.

If the sequential dependent segment that is the target of the POS command is inserted in the same synchronization interval, no position information is returned. Bytes 11-18 contain X'FF'; other fields contain normal data.

Unused CIs
A 4-byte field containing the number of unused CIs in the sequential dependent part.

Unused CIs
A 4-byte field containing the number of unused CIs in the independent overflow part.

Locating the last inserted sequential dependent segment

You can also retrieve the position information for the most recently inserted sequential dependent segment of a given root segment.

To do this, you issue a POS command with a qualification statement containing the root segment as the segment name. The current position after this type of command follows the same rules as position after a GU.

After a successful command, the I/O area contains:

LL A 2-byte field containing the total length of the data in the I/O area, in binary.

Area Name
An 8-byte field giving the ddname from the AREA statement.

Position
An 8-byte field containing the position information for the most recently inserted sequential dependent segment. This field contains zeros provided no sequential dependent for this root exist.

Unused CIs
A 4-byte field containing the number of unused CIs in the sequential dependent part.

Unused CIs
A 4-byte field containing the number of unused CIs in the independent overflow part.

Identifying free space with the POS command

To retrieve the area name and the next available position within the sequential dependent part from all online areas, you can issue an unqualified POS command. This type of command also retrieves the free space in the independent overflow and sequential dependent parts.

After a successful unqualified POS command, the I/O area contains the length (LL) followed by the same number of entries as areas within the database. Each entry contains field two through five shown below:

LL A 2-byte field containing the total length of the data in the I/O area, in binary. The length includes the 2 bytes for the LL field, plus 24 bytes for each entry.

Area Name
An 8-byte field giving the ddname from the AREA statement.

Position
An 8-byte field giving the next available position within the sequential dependent part.

Unused CIs
A 4-byte field containing the number of unused CIs in the sequential dependent part.

Unused CIs
A 4-byte field containing the number of unused CIs in the independent overflow part.

The P processing option

If the P processing option has been specified (with the PROCOPT parameter) in the PCB for your program, a GC status code is returned to your program whenever a command to retrieve or insert a segment causes a Unit of Work (UOW) boundary to be crossed.

Although crossing the UOW boundary probably has no particular significance for your program, the GC status code indicates that this is a good time to issue a CHKP command. The advantages of doing this are:

- Your position in the database is kept. Issuing a CHKP normally causes position in the database to be lost, and the application program has to reestablish position before it can resume processing.
- Commit points occur at regular intervals.

When a GC status code is returned, no data is retrieved or inserted. In your program, you can either:

- Issue a CHKP command, and resume database processing by reissuing the command that caused the GC status code.
- Ignore the GC status code and resume database processing by reissuing the command that caused the status code.

Chapter 33. Comparing command-level and call-level programs

Call-level and command-level programs exhibit different behavior.

DL/I calls for IMS and CICS

The following table provides a reference for using DL/I calls in a batch, batch-oriented BMP, or CICS with DBCTL environment.

Table 88. DL/I calls available to IMS and CICS command-level application programs.

Request type	Batch	Batch-oriented BMP	CICS with DBCTL[1]
CHKP call (symbolic)	Yes	Yes	No
CHKP call (basic)	Yes	Yes	No
GSCD call[2]	Yes	No	No
INIT call	Yes	Yes	Yes
ISRT call (initial load)	Yes	No	No
ISRT call	Yes	Yes	Yes
LOG call	Yes	Yes	Yes
SCHD call	No	No	Yes
ROLB call	Yes	Yes	No
ROLL call	Yes	Yes	No
ROLS call (Roll Back to SETS)[3]	Yes	Yes	Yes
ROLS call (Roll Back to Commit)	Yes	Yes	Yes
SETS call[3]	Yes	Yes	Yes
STAT call[4]	Yes	Yes	Yes
TERM call	No	No	Yes
XRST call	Yes	Yes	No

1. In a CICS remote DL/I environment, CALLs in the CICS-DBCTL column are supported if you are shipping a function to a remote CICS that uses DBCTL.
2. GSCD is a Product-sensitive Programming Interface.
3. SETS and ROLS calls are not valid when the PSB contains a DEDB.
4. STAT is a Product-sensitive Programming Interface.

Comparing EXEC DLI commands and DL/I calls

Use the appropriate EXEC DLI commands and DL/I calls in your program.

The following table compares EXEC DLI commands with DL/I calls. For example, in a command-level program, you use the `LOAD` command instead of the `ISRT` call to initially load a database.

© Copyright IBM Corp. 1974, 2015

Table 89. Comparing call-level and command-level programs: commands and calls.

Call-level	Command-level	Purpose
INIT call	ACCEPT command	Initialize for data availability status codes.
CHKP call (basic)	CHKP command	Issue a basic checkpoint.
DEQ call	DEQ command	Release segments retrieved using LOCKCLASS option or Q command code.
DLET call	DLET command	Delete segments from a database.
GU, GN, and GNP calls	GU, GN, and GNP commands[1]	Retrieve segments from a database.
GHU, GHN, and GHNP calls[1]	GU, GN, and GNP commands[1]	Retrieve segments from a database for updating.
GSCD call	GSCD call[2]	Retrieve system addresses.
ISRT call	ISRT command	Add segments to a database.
ISRT call	LOAD command	Initially load a database.
LOG call	LOG command	Write a message to the system log.
POS call	POS command	Retrieve positioning or space usage or positioning and space usage in a DEDB area.
INIT call	ACCEPT command	Initialize for data availability status.
INIT call	QUERY command	Obtain information of initial data availability.
INIT call	REFRESH command	Availability information after using a PCB.
REPL call	REPL command	Replace segments in a database.
XRST call	RETRIEVE command	Issue an extended restart.
ROLL or ROLB call	ROLL or ROLB command	Dynamically back out changes.
ROLS call	ROLS command	Back out to a previously set backout point.
PCB call	SCHD command	Schedule a PSB.
SETS call	SETS command	Set a backout point.
SETU call	SETU command	Set a backout point even if unsupported PCBs (like DEDBs or MSDBs) are present.
STAT call[3]	STAT command	Obtain system and buffer pool statistics.
CHKP call (extended)	SYMCHKP command	Issue a symbolic checkpoint.
TERM call	TERM command	Terminate a PSB.
XRST call	XRST command	Issue an extended restart.

Notes:

1. Get commands are just like Get Hold calls, and the performance of Get commands and Get calls is the same.
2. You can use the GSCD call in a batch command-level program. GSCD is a Product-sensitive Programming Interface.
3. STAT is a Product-sensitive Programming Interface.

Comparing command codes and options

The following table compares the options you use with EXEC DLI commands with the command codes you use with DL/I calls. For example, the LOCKED option performs the same function as a Q command code.

Table 90. Comparing call-level and command-level programs: command codes and options

Call- Level	Command-Level	Allows You to . . .
C	KEYS option	Use the concatenated key of a segment to identify the segment.
D	INTO or FROM specified on segment level to be retrieved or inserted.	Retrieve or insert a sequence of segments in a hierarchic path using only one request, instead of having to use a separate request for each segment. (Path call or command).
F	FIRST option	Back up to the first occurrence of a segment under its parent when searching for a particular segment occurrence. Disregarded for a root segment.
L	LAST option	Retrieve the last occurrence of a segment under its parent.
M	MOVENEXT option	Set a subset pointer to the segment following the current segment.
N	Leave out the SEGMENT option for segments you do not want replaced.	Designate segments you do not want replaced, when replacing segments after a get hold request. Usually used when replacing a path of segments.
P	SETPARENT	Set parentage at a higher level than what it usually is (the lowest hierarchic level of the request).
Q	LOCKCLASS, LOCKED	Reserve a segment so that other programs are not able to update it until you have finished processing it.
R	GETFIRST option	Retrieve the first segment in a subset.
S	SET option	Unconditionally set a subset pointer to the current segment.
U	No equivalent for command level programs.	Limit the search for a segment to the dependents of the segment occurrence on which position is established.
V	CURRENT option	Use the hierarchic level of and levels above the current position as qualifications for the segment.
W	SETCOND option	Conditionally set a subset pointer to the current segment.
Z	SETZERO option	Set a subset pointer to zero.
–	No command-level equivalent.	Null. Use an SSA in command code format without specifying the command code. Can be replaced during execution with the command codes you want.

Chapter 34. Data availability enhancements

Your program might fail when it receives a status code indicating that a DL/I full-function database is unavailable. To avoid this, you can use these data availability enhancements. After a PSB has been scheduled in DBCTL, your application program can issue requests to indicate to IMS that the program can handle data availability status codes and to obtain information about the availability of each database.

Accepting database availability status codes

These status codes occur because PSB scheduling was completed without all of the referenced databases being available. Use the ACCEPT command to tell DBCTL to return a status code instead of abending the program:

```
EXEC DLI ACCEPT STATUSGROUP('A');
```

Obtaining information about database availability

You can put data availability status codes into each of the DB PCBs if:

- In a CICS DBCTL environment, by using the PSB scheduling request command, SCHD.
- In a Batch or BMP environment, at initialization time.

You can obtain the data availability status codes within the DL/I interface block (DIB) by using the following QUERY command:

```
EXEC DLI QUERY USING PCB(n);
```

n specifies the PCB.

The QUERY command is used after scheduling the PSB but before making the first database call. If the program has already issued a call using a DB PCB, then the QUERY command must follow the REFRESH command:

```
EXEC DLI REFRESH DBQUERY
```

The REFRESH command updates the information in the DIB. You can only issue this command one time.

For full-function databases, the DIBSTAT should contain NA, NU, TH, or blanks. For MSDBs and DEDBs, the DIBSTAT always contains blanks.

If a CICS command language translator has been used to translate the EXEC DLI commands, then, in addition to data availability status, the DBDNAME will be returned in the DIB field DIBDBDNM. Also, the name of the database organization will be returned in the DIB field DIBDBORG.

© Copyright IBM Corp. 1974, 2015

Part 5. Java application development for IMS

IMS provides support for developing applications using the Java programming language.

© Copyright IBM Corp. 1974, 2015

Chapter 35. IMS solutions for Java development overview

You can write Java applications to access IMS databases and process IMS transactions by using the drivers and resource adapters of the IMS solutions for Java development.

The IMS solutions for Java development include the IMS Universal drivers, the IMS Java dependent region resource adapter, and the classic Java APIs for IMS.

IMS Universal drivers

The IMS Universal drivers are a set of SMP/E-installable Java drivers and resource adapters that enable access to IMS from z/OS and distributed (non-z/OS) platforms. The IMS Universal drivers are built on industry standards and open specifications. Two types of connectivity are supported by the IMS Universal drivers: local connectivity to IMS databases on the same LPAR (*type-2 connectivity*) and distributed connectivity through TCP/IP (*type-4 connectivity*). Java applications that use the type-2 IMS Universal drivers must reside on the same logical partition (LPAR) as the IMS subsystem. Java applications that use the type-4 IMS Universal drivers can reside on the same logical partition (LPAR) or on a different LPAR from the IMS subsystem.

The IMS Universal drivers enable access to IMS from multiple environments, including:

- WebSphere Application Server for z/OS
- CICS Transaction Server for z/OS
- IMS on the host in JMP and JBP regions

The IMS Universal drivers include:

- IMS Universal Database resource adapter: A Java EE Connector Architecture (JCA) 1.5-compliant resource adapter
- IMS Universal JDBC driver: A Java Database Connectivity (JDBC) driver that implements the JDBC 3.0 API
- IMS Universal DL/I driver: A Java API for making calls with traditional DL/I programming semantics

IMS Java dependent region resource adapter

The IMS Java dependent region resource adapter is a set of Java classes and interfaces that support IMS database access and IMS message queue processing within Java batch processing (JBP) and Java message processing (JMP) regions. The IMS Java dependent region resource adapter provides Java application programs running in JMP or JBP regions with similar DL/I functionality to that provided in message processing program (MPP) and non-message driven BMP regions, such as:

- Accessing IMS message queues to read and write messages
- Performing program switches
- Commit and rollback processing
- Accessing IMS databases in an IMS DB/TM environment
- Accessing GSAM databases in IMS DB/TM and DCCTL environments
- Database recovery (CHKP/XRST)

© Copyright IBM Corp. 1974, 2015

Classic Java APIs for IMS

The classic Java APIs for IMS are a set of SMP/E-installable Java classes and interfaces to access IMS from multiple runtime environments.

The classic Java APIs for IMS are delivered with IMS Version 10 and earlier, but are not supported in IMS Version 11 and later.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

The classic Java APIs for IMS includes JCA 1.0 support, and a IMS classic JDBC driver based on the JDBC 2.1 standard for issuing SQL queries to IMS databases.

The classic Java APIs for IMS and the IMS Universal drivers for type-2 connectivity are built on top of existing assembler interfaces to IMS. For the IMS environment, the CEETDLI interface is used. DB2 for z/OS and WebSphere Application Server use the AERTDLI interface. CICS uses the AIBTDLI interface. The classic Java APIs for IMS and the IMS Universal drivers for type-2 connectivity detect which environment is used, which enables the APIs to use the appropriate assembler interface to IMS at run time. The ability for the APIs to detect environment is transparent to the application.

IMS DB resource adapter

IMS provides the IMS DB resource adapter as a JCA resource adaptor that is built on the classic Java APIs for IMS for deployment on WebSphere Application Server. IMS DB distributed resource adapter for WebSphere Application Server for z/OS is installed using SMP/E with IMS. This resource adapter provides a set of Java class libraries that you can use to write Java EE application programs that access IMS databases from WebSphere Application Server for z/OS.

Recommendation: Because the IMS Universal Database resource adapter is built on industry standards and open specifications, and provides more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal Database resource adapter to develop Java EE applications that access IMS from WebSphere Application Server.

Related concepts:

Chapter 38, “Programming Java dependent regions,” on page 641

IMS Explorer for Development overview

Chapter 11, “Writing your application programs for IMS DB,” on page 181

“Formatting DL/I calls for language interfaces” on page 213

Related tasks:

Configuring external Java environment connections (Communications and Connections)

Related reference:

“Coding a program in Java” on page 205

“Java application programming for IMS” on page 221

Chapter 37, “Programming with the IMS Universal drivers,” on page 549

Chapter 39, “Programming with the classic Java APIs for IMS,” on page 669

Software requirements for Java applications that access IMS databases (Release Planning)

Java API documentation (Javadoc) (Application Programming APIs)

Java API specification for the classic Java APIs for IMS (IMS V11 Application programming APIs)

Chapter 36. Comparison of hierarchical and relational databases

The following information describes the differences between the hierarchical model for IMS databases and the standard relational database model.

A database segment definition defines the fields for a set of segment instances similar to the way a relational table defines columns for a set of rows in a table. In this way, segments relate to relational tables, and fields in a segment relate to columns in a relational table.

The name of an IMS segment becomes the table name in an SQL query, and the name of a field becomes the column name in the SQL query.

A fundamental difference between segments in a hierarchical database and tables in a relational database is that, in a hierarchical database, segments are implicitly joined with each other. In a relational database, you must explicitly join two tables. A segment instance in a hierarchical database is already joined with its parent segment and its child segments, which are all along the same hierarchical path. In a relational database, this relationship between tables is captured by foreign keys and primary keys.

This section compares the Dealership sample database, to a relational representation of the database. The Dealership sample DBDs are available with the IMS Enterprise Suite Explorer for Development, in the `<installation location>\IMS Explorer samples` directory.

Important: This information provides only a comparison between relational and hierarchical databases.

The Dealership sample database contains five segment types, which are shown in the following figure. The root segment is the Dealer segment. Under the Dealer segment is its child segment, the Model segment. Under the Model segment are its children: the segments Order, Sales, and Stock.

The following figure shows the structure and each segment of the Dealership sample database.

© Copyright IBM Corp. 1974, 2015

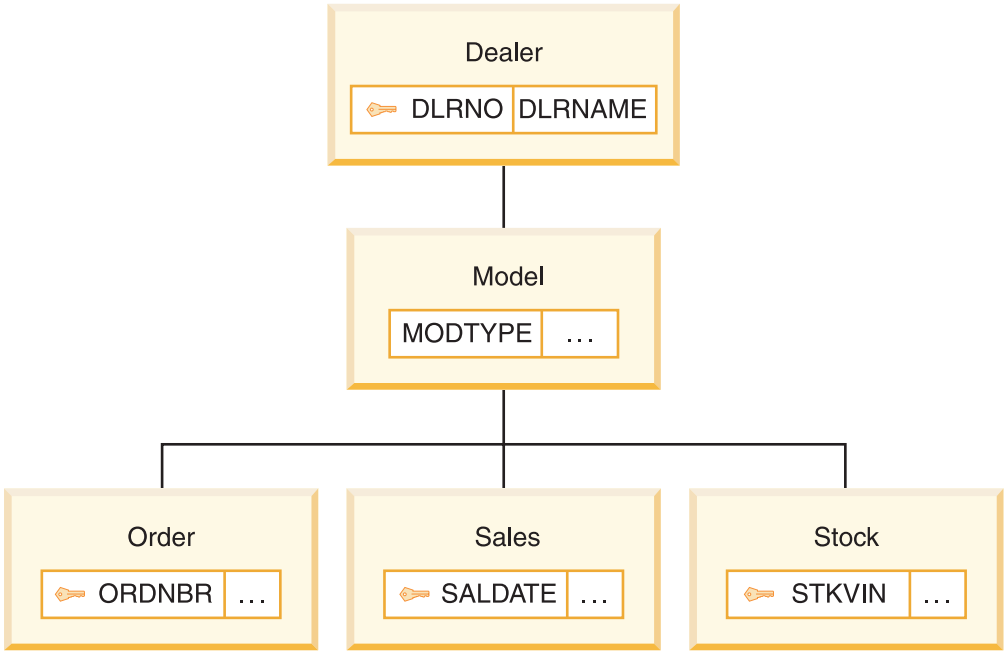


Figure 98. Segments of the Dealership sample database

The Dealer segment identifies a dealer that sells cars. The segment contains a dealer name in the field DLRNAME, and a unique dealer number in the field DLRNO.

Dealers carry car types, each of which has a corresponding Model segment. A Model segment contains a type code in the field MODTYPE.

Each car that is ordered for the dealership has an Order segment. A Stock segment is created for each car that is available for sale in the dealer's inventory. When the car is sold, a Sales segment is created.

The following figure shows a relational representation of the IMS database record shown in Figure 98.

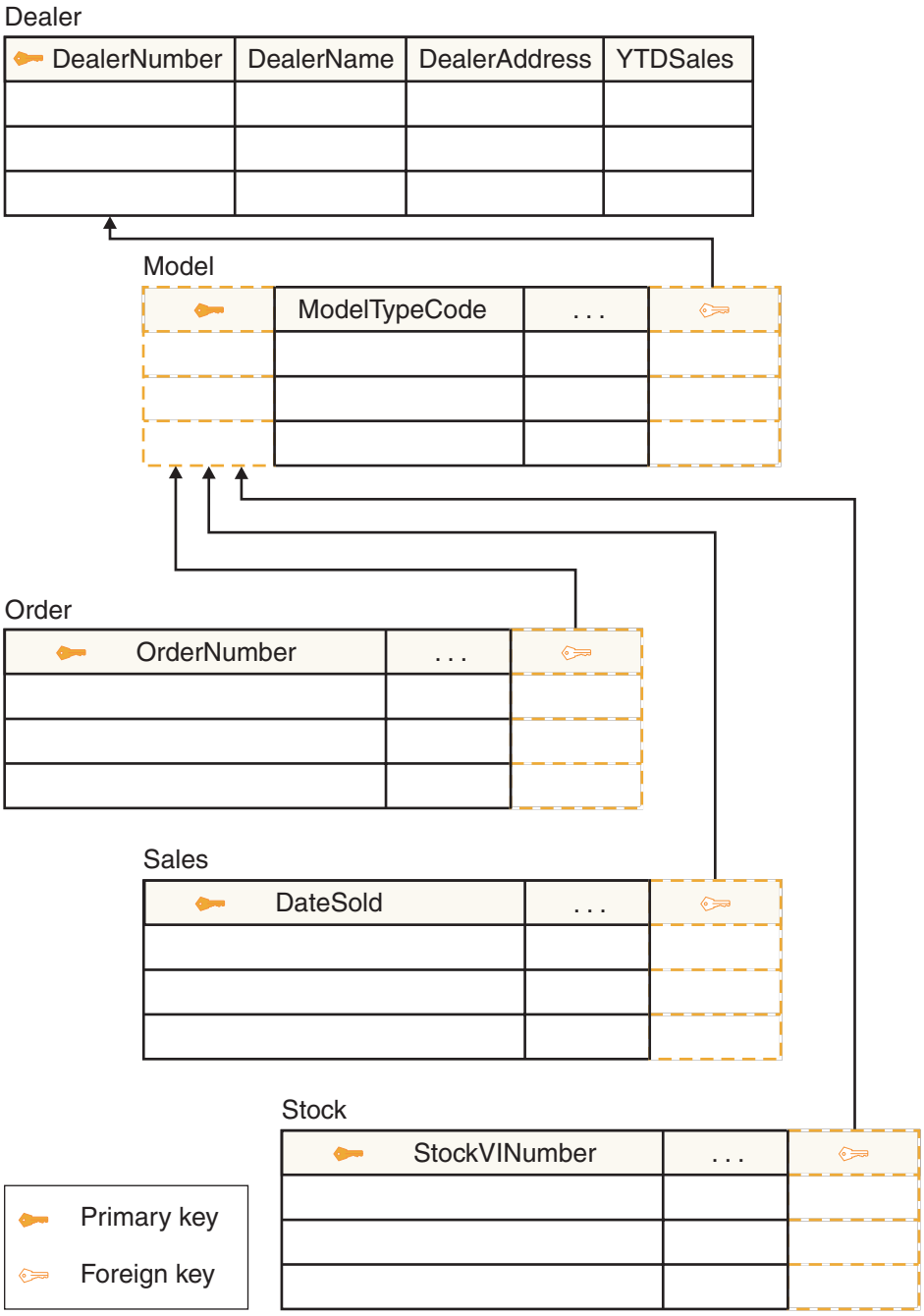


Figure 99. Relational representation of the Dealership sample database

If a segment does not have a unique key, which is similar to a primary key in relational databases, view the corresponding relational table as having a generated primary key added to its column (field) list. An example of a generated primary key is in the Model table (segment) of the figure above. Similar to referential integrity in relational databases, you cannot insert, for example, an Order (child) segment to the database without it being a child of a specific Model (parent) segment.

Also note that the field (column) names have been renamed. You can rename segments and fields to more meaningful names by using the IMS Explorer for Development.

An occurrence of a segment in a hierarchical database corresponds to a row (or tuple) of a table in a relational database.

The following figure shows three Dealership database records.

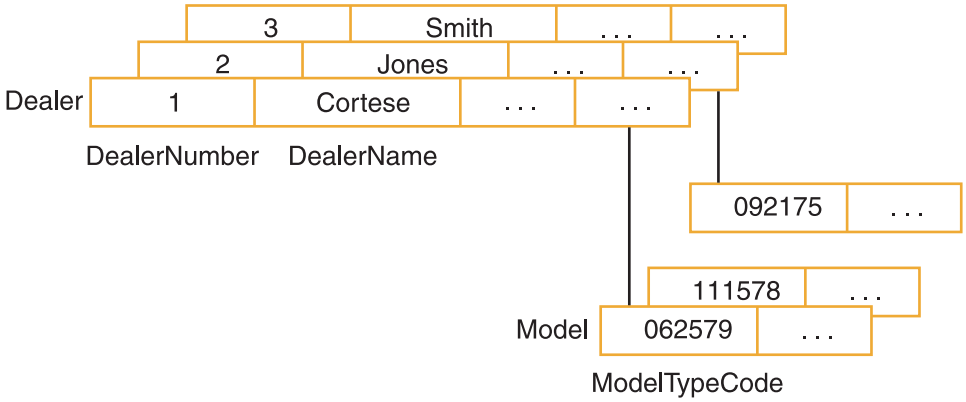


Figure 100. Segment occurrences in the Dealership sample database

The Dealer segment occurrences have dependent Model segment occurrences.

The following figure shows the relational representation of the dependent model segment occurrences.

Dealer

DealerNumber	DealerName	. . .	. . .
1	Cortese		
2	Jones		
3	Smith		

Model

ModelTypeCode	. . .	
062579		1
111578		1
092175		2

Figure 101. Relational representation of segment occurrences in the Dealership database

In the following example that shows the SELECT statement of an SQL call, Model is a segment name that is used as a table name in the query:

```
SELECT * FROM Model
```

The following example, ModelTypeCode is the name of a field that is contained in the Model segment and it is used in the SQL query as a column name:

```
SELECT * FROM Model WHERE ModelTypeCode = '062579'
```

In the two preceding examples, Model and ModelTypeCode are alias names that are assigned by using the DLIModel utility. These names will not likely be the same 8-character names that are used in the database description (DBD) for IMS. Alias names act as references to the 8-character names that are described in the DBD.

Chapter 37. Programming with the IMS Universal drivers

Use these topics to design, write, and maintain application programs for IMS Version 11 using the IMS Universal drivers.

Related concepts:

Chapter 35, "IMS solutions for Java development overview," on page 539

"Overview of the IMS Java dependent regions" on page 641

Related reference:

Software requirements for Java application programs that use the IMS Universal drivers (Release Planning)

IMS Universal drivers overview

The IMS Universal drivers are software components that provide Java applications with connectivity and access to IMS databases from z/OS and distributed environments through TCP/IP. Java applications that use the type-2 IMS Universal drivers must reside on the same logical partition (LPAR) as the IMS subsystem. Java applications that use the type-4 IMS Universal drivers can reside on the same logical partition (LPAR) or on a different LPAR from the IMS subsystem.

Programming approaches

The IMS Universal drivers provide an application programming framework that offers multiple options for access to IMS data. These programming options include:

IMS Universal Database resource adapter
Provides connectivity to IMS databases from a Java Platform, Enterprise Edition (Java EE) environment, and access to IMS data using the Common Client Interface (CCI) and Java Database Connectivity (JDBC) interfaces.

IMS Universal JDBC driver
Provides a stand-alone JDBC 3.0 driver for making SQL-based database calls to IMS databases.

IMS Universal DL/I driver
Provides a stand-alone Java API for writing granular queries to IMS databases using programming semantics similar to traditional DL/I calls.

Open standards

The IMS Universal drivers are built on the following industry open standards and interfaces:

Java EE Connector Architecture (JCA)
JCA is the Java standard for connecting Enterprise Information Systems (EISs) such as IMS into the Java EE framework. Using JCA, you can simplify application development and take advantage of the services that can be provided by a Java EE application server, such as connection management, transaction management, and security management. The Common Client Interface (CCI) is the interface in JCA that provides access from Java EE clients, such as Enterprise JavaBeans (EJB) applications, JavaServer Pages (JSP), and Java servlets, to backend IMS subsystems.

© Copyright IBM Corp. 1974, 2015

Java Database Connectivity (JDBC)
JDBC is the SQL-based standard interface for database access. It is the industry standard for database-independent connectivity between the Java programming language and any database that has implemented the JDBC interface.

Distributed Relational Database Architecture (DRDA) specification
DRDA is an open architecture that enables communication between applications and database systems on disparate platforms. These applications and database systems can be provided by different vendors and the platforms can be different hardware and software architectures. DRDA provides distributed database access with built-in support for distributed, two-phase commit transactions.

Related reference:

DRDA DDM command architecture reference (Application Programming APIs)

Software requirements for Java application programs that use the IMS Universal drivers (Release Planning)

Distributed and local connectivity with the IMS Universal drivers

The IMS Universal drivers support distributed (*type-4*) and local (*type-2*) connectivity to IMS databases.

Distributed connectivity with the type-4 IMS Universal drivers

With type-4 connectivity, the IMS Universal drivers can run on any platform that supports TCP/IP and a Java Virtual Machine (JVM), including z/OS. To access IMS databases, the type-4 IMS Universal drivers first establish a TCP/IP-based socket connection to IMS Connect. IMS Connect is responsible for routing the request to the IMS databases using the Open Database Manager (ODBM), and sending the response back to the client application. The DRDA protocol is used internally in the implementation of the type-4 IMS Universal drivers. You do not need to know DRDA to use the type-4 IMS Universal drivers.

The type-4 IMS Universal drivers support two-phase commit (XA) transactions. IMS Connect builds the necessary z/OS Resource Recovery Services (RRS) structure to support the two-phase commit protocol. If two-phase commit transactions are not used, RRS is not required.

When establishing a connection to IMS, the **driverType** connection property must be set to indicate distributed (*type-4*) connectivity to IMS

After successful authentication, the IMS Universal drivers sends other socket connection information, such as program specification block (PSB) name and IMS database subsystem, to IMS Connect and ODBM in order to allocate the PSB to connect to the database.

A connection to an IMS database is established only when a program specification block (PSB) is allocated. Authorization for a particular PSB is done by the ODBM component during the allocation of a PSB.

The type-4 IMS Universal drivers support connection pooling, which limits the time that is needed for allocation and deallocation of TCP/IP socket connections. To maximize connection reuse, only the socket attributes of a connection are pooled. These attributes include the IP address and port number that the host IMS Connect is listening on. As a result, the physical socket connection can be reused and additional attributes can be sent on this socket in order to connect to an IMS database. When a client application of the type-4 IMS Universal drivers makes a connection to IMS, this means:

- A one-to-one relationship is established between a client socket and an allocated PSB that contains one or more IMS databases.
- A one-to-many relationship is established between IMS Connect and the possible number of database connections it can handle at one time.
- IMS Connect does the user authentication.
- ODBM ensures that the authenticated user is authorized to access the given PSB.

The following figure shows how the type-4 IMS Universal drivers route communications between your Java client applications running in a distributed environment and an IMS subsystem.

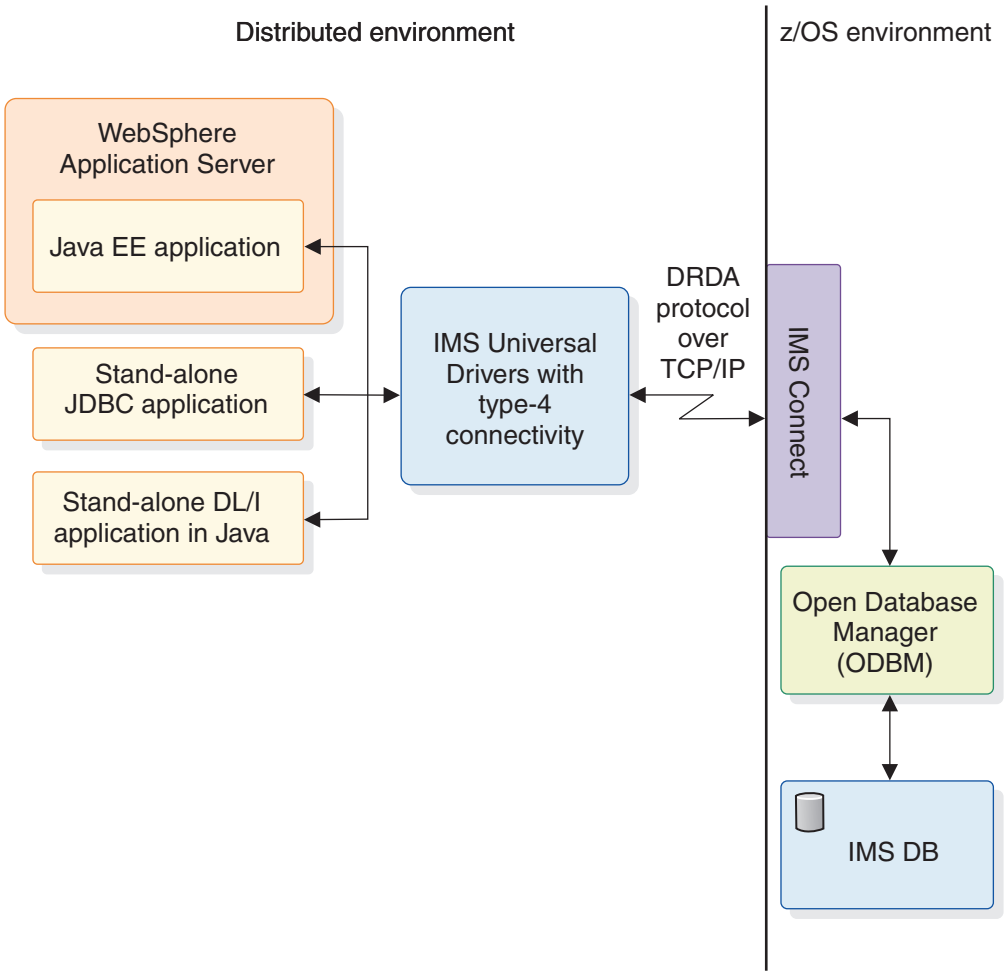


Figure 102. Distributed connectivity with the type-4 IMS Universal drivers

You can also use the type-4 IMS Universal drivers if your Java clients are running in a z/OS environment but are located on a separate logical partition from the IMS subsystem. Use type-4 connectivity from a z/OS environment if you want to

isolate the application runtime environment from the IMS subsystem environment.

Local connectivity with the type-2 IMS Universal drivers

Local connectivity with the type-2 IMS Universal drivers is targeted for the z/OS platform and runtime environments. Use type-2 connectivity when connecting to IMS subsystems in the same logical partition (LPAR).

The following table shows the z/OS runtime environments that support client applications of the type-2 IMS Universal drivers.

Table 91. z/OS runtime environment support for the type-2 IMS Universal drivers

z/OS runtime environment	Type-2 IMS Universal drivers supported
WebSphere Application Server for z/OS	• IMS Universal Database resource adapter
IMS Java dependent regions (JMP and JBP regions); CICS; USS	• IMS Universal DL/I driver • IMS Universal JDBC driver

Because it runs on the same LPAR as the IMS subsystem, during connection time, a client application of the type-2 IMS Universal drivers does not need to supply an IP address, port number, user ID, or password. The **driverType** property must be set to indicate local (*type*-2) connectivity to IMS.

The following figure shows how the type-2 IMS Universal drivers route communications between your Java client applications running in an LPAR inside a z/OS mainframe environment and an IMS subsystem located in the same LPAR.

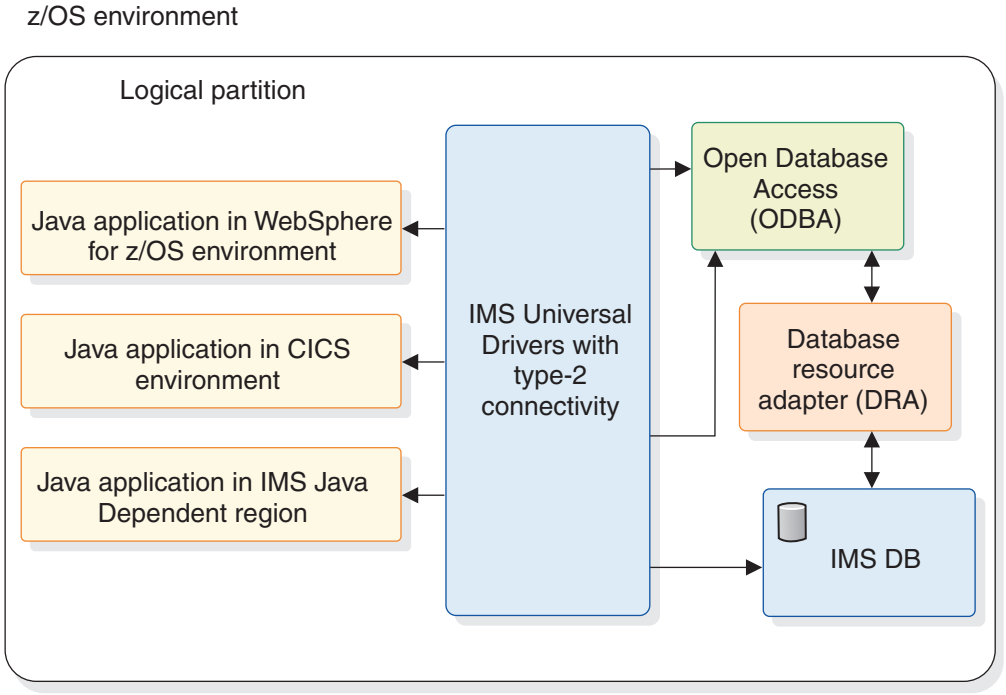


Figure 103. Local connectivity with the type-2 IMS Universal drivers

RRSLocalOption connectivity type

In addition to type-4 and type-2 connectivity, the RRSLocalOption connectivity type is supported by the IMS Universal Database resource adapter running on WebSphere Application Server for z/OS. With RRSLocalOption connectivity, applications using the IMS Universal Database resource adapter do not issue commit or rollback calls. Instead, transaction processing is managed by WebSphere Application Server for z/OS. Two-phase commit (XA) transaction processing is not supported with RRSLocalOption connectivity type.

Related concepts:

"Accessing IMS data from WebSphere Application Server for z/OS with the classic Java APIs for IMS" on page 669

"Accessing IMS data from DB2 for z/OS stored procedures using the classic Java APIs for IMS" on page 675

"Accessing IMS data from CICS with the classic Java APIs for IMS" on page 676

IMS Connect support for access to IMS DB (Communications and Connections)

CSL ODBM administration (System Administration)

Comparison of IMS Universal drivers programming approaches for accessing IMS

Depending on your IT infrastructure, solution architecture, and application design, choose the IMS Universal drivers programming approach that is best for your development scenario.

The following table lists the recommended IMS Universal drivers programming approach to use, based on the application programmer's choice of application platform, data access method, and transaction processing option.

Table 92. Comparison of programming approaches for accessing IMS

Application platform	Data access method	Transaction processing required	Recommended approach
WebSphere Application Server for distributed platforms or WebSphere Application Server for z/OS	CCI programming interface to perform SQL or DL/I data operations.	Local transaction processing only.	Use the IMS Universal Database resource adapter with local transaction support (imsudbLocal.rar), and make SQL calls with the SQLInteractionSpec class or DL/I calls with the DLIInteractionSpec class.
	CCI programming interface to perform SQL or DL/I data operations.	Two-phase (XA) commit processing[1] or local transaction processing.	Use the IMS Universal Database resource adapter with XA transaction support (imsudbXA.rar), and make SQL calls with the SQLInteractionSpec class or DL/I calls with the DLIInteractionSpec class.
	JDBC programming interface to perform SQL data operations.	Local transaction processing only.	Use the IMS Universal JCA/JDBC driver version of the IMS Universal Database resource adapter with local transaction support (imsudbJLocal.rar), and make SQL calls with the JDBC API.
	JDBC programming interface to perform SQL data operations.	Two-phase (XA) commit processing[1] or local transaction processing.	Use the IMS Universal JCA/JDBC driver version of the IMS Universal Database resource adapter with XA transaction support (imsudbJXA.rar), and make SQL calls with the JDBC API.

Table 92. Comparison of programming approaches for accessing IMS (continued)

Application platform	Data access method	Transaction processing required	Recommended approach
Standalone Java application (outside a Java EE application server) that resides on a distributed platform or a z/OS platform	JDBC programming interface to perform SQL data operations.	Two-phase (XA) commit processing[2] or local transaction processing.	Use the IMS Universal JDBC driver (imsudb.jar), and make SQL calls with the JDBC API.
	Traditional DL/I programming semantics to perform data operations.	Two-phase (XA) commit processing[2] or local transaction processing.	Use the IMS Universal DL/I driver (imsudb.jar), and make DL/I calls with the PCB class.
Standalone non-Java application that resides on a distributed platform or a z/OS platform	Data access using DRDA protocol.	Two-phase (XA) commit processing or local transaction processing.	Use a programming language of your choice to issue DDM commands to IMS Connect. The application programmer is responsible for implementing the two-phase commit mechanism.

Note:

1. XA transaction support is available only with type-4 connectivity.
2. The driver is enabled for local and XA transactions, but the application programmer is responsible for implementing the two-phase commit mechanism. XA transaction support is available only with type-4 connectivity.

Support for variable-length database segments with the IMS Universal drivers

The IMS Universal database resource adapter and the IMS Universal JDBC driver manage variable-length segments on behalf of client application programs. Application programs that use the IMS Universal DL/I driver must manage the LL field data for variable-length segments.

SQL language conventions assume that the target database is relational. Relational database managers do not use the concept of *variable-length* data structures that are managed by an external application program. Because IMS is a hierarchical database, the IMS Universal database resource adapter and IMS Universal JDBC driver translate SQL statements into DL/I calls that can be interpreted by the IMS Database Manager. When using the IMS Database Manager, applications must manage the length of individual variable-length segment instances with the LL field for the segment. Applications that use the IMS Universal DL/I driver are responsible for managing the LL field.

However, application programs that use the IMS Universal database resource adapter or the IMS Universal JDBC driver treat IMS databases as standard JDBC data sources. The IMS Universal database resource adapter and IMS Universal JDBC driver internally manage the LL field on behalf of the application, so that the application program does not need to manage the segment length or the size of the I/O area. For read operations, the IMS Universal database resource adapter and

IMS Universal JDBC driver handle the offsets and lengths of all the segments and fields returned. By default, the SQL result set does not contain the LL field information. For update or insert operations, each instance of a variable-length segment is automatically expanded to contain the largest field (determined by the field length and offset) in the segment instance.

In a variable-length segment, some fields might be nullable. In IMS, a nullable field is a field that has a starting offset or combined offset and length larger than the minimum length of the segment. You can determine if a nullable field exists for a particular segment instance by comparing the LL value for the instance to the combined offset and length for the nullable field. If the LL value is less than the combined offset and length of the field, the field is null. For example, if a segment definition includes a field that starts at offset 50 and is length 5, it is nullable if the minimum length of the segment is less than 55. It is null for a particular segment instance if the LL value for that instance is less than 55.

Using the LL field with the IMS Universal database resource adapter and IMS Universal JDBC driver

By default, the LL field for a variable length segment is not returned as a visible column for SQL queries. When the LL field is not requested, the IMS Universal database resource adapter and IMS Universal JDBC driver manage the LL field on behalf of the application program. The LL field data is not accessible by any type of query (including `SELECT *`) unless your application program explicitly requests the column when it creates a data connection to IMS.

If you want to manage the LL field at the application level, and your application uses the IMS Universal database resource adapter or the IMS Universal JDBC driver, you must explicitly request the LL field data by setting the llField property to `true`.

Your application can set the llField property to `true` in the standard properties list of either of the following interfaces:

```
java.sql.DriverManager.getConnection(String url, Properties properties)
com.ibm.ims.jdbc.Datasource.setProperties(Properties properties)
```

When the llField=true property is set, the LL field is exposed as a normal column in the standard SQL result set for all operations. You can read, insert, or update the LL field data directly. Deleting the LL field data also deletes the rest of the associated database record. To set a field to the null state, set the length of the segment (the value of the LL field column) to be smaller than the offset of the field within the segment.

The LL field is 2 bytes long and must be handled as BINARY, SHORT, or USHORT data.

You can also use the java.sql.ResultSet.wasNull method to determine whether a nullable field exists in an instance of a variable-length segment without examining the LL data.

Checking for null field instances with the IMS Universal DL/I driver

Applications that use the IMS Universal DL/I driver always receive the LL field data for a variable-length segment. You can determine if a field is null in a

segment instance in one of two ways: either compare the LL field data to the offset of the field, or use the com.ibm.ims.dli.Path.wasNull() method.

The com.ibm.ims.dli.Path.wasNull() method returns a boolean value for the null state of the last field that was read. The returned value is `true` if the field is null. You must attempt to read a field before calling the wasNull() method to determine whether the field is null.

Related concepts:

Variable-length segments (Database Administration)

Related tasks:

How to specify variable-length segments (Database Administration)

Generating the runtime Java metadata class

To connect to an IMS database using the IMS Universal drivers or the IMS classic JDBC driver provided by the classic Java APIs for IMS, you need to include on your Java classpath the Java metadata class that provides the database view.

The Java metadata class is generated using the IMS Enterprise Suite Explorer for Development (or the IMS Enterprise Suite DLIModel utility plug-in for theIMS classic JDBC driver) . The Java metadata class represents the application view information specified by a program specification block (PSB) and its related Program Control Blocks (PCBs). The Java metadata class provides a one-to-one mapping to the segments and fields defined in the PSB.

To generate the metadata class, use the IMS Explorer for Development to import the application PSB source and related DBD source files. Optionally, you can also import COBOL copybooks and PL/I INCLUDE files. The Java metadata class must be compiled and made available through the classpath for any Java application attempting to access IMS data using that PSB.

During database connection setup, pass the name of this metadata class to the resource adapter or JDBC driver. The Java metadata class is used at runtime by the IMS Universal drivers and the IMS classic JDBC driver to process both SQL and Java-based DL/I calls.

The default segment encoding of the database metadata class produced by the IMS Explorer for Development is `cp1047`. To change the segment encoding, use the com.ibm.ims.base.DLIBaseSegment.setDefaultEncoding method.

Related concepts:

IMS Explorer for Development overview

Hospital database example

The code examples for the IMS Universal drivers application programming topics use the Hospital database.

The following figure shows the hierarchical structure of the segments in the Hospital database.

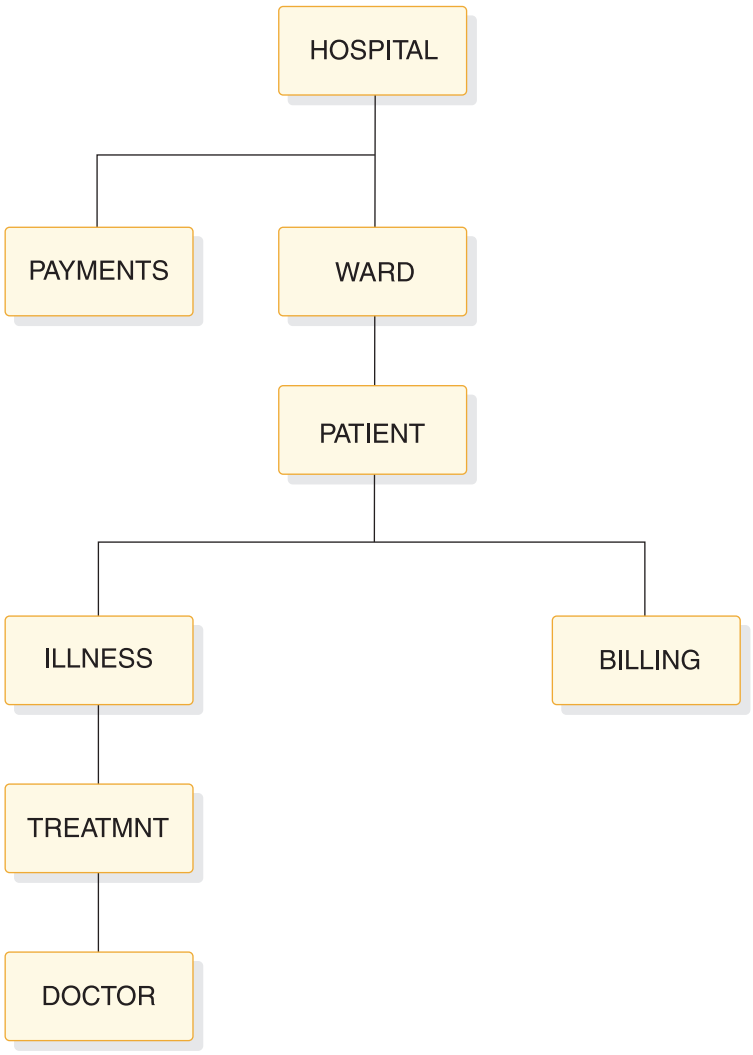


Figure 104. Segments of the Hospital database

Each node in the figure represents a segment:

- The HOSPITAL segment is the root segment in the database.
- PAYMENTS and WARD are child segments of the HOSPITAL segment.
- WARD has a direct descendent segment named PATIENT.
- ILLNESS and BILLING are the child segments of the PATIENT.
- ILLNESS has a child segment named TREATMENT that stores details about patient treatment.
- The child segment of ILLNESS, DOCTOR, is the lowest level segment in the database hierarchy.

The tables that follow show the layouts of each segment in the Hospital database.

HOSPITAL segment

The following table shows the HOSPITAL segment, which has two fields:

- The hospital code (HOSPCODE)
- The hospital name (HOSPNAME)

HOSPCODE is a unique key field.

Field name	Field length (in bytes)
HOSPCODE	12
HOSPNAME	17

PAYMENTS segment

The following table shows the PAYMENTS segment, which has two fields:

- The patient number (PATNUM)
- The payment amount (AMOUNT)

Field name	Field length (in bytes)
PATNUM	4
AMOUNT	8

WARD segment

The following table shows the WARD segment, which has five fields:

- The ward number (WARDNO)
- The ward name (WARDNAME)
- The patient count (PATCOUNT)
- The nurse count (NURCOUNT)
- The doctor count (DOCCOUNT)

WARDNO is a unique key field.

Field name	Field length (in bytes)
WARDNO	2
WARDNAME	4
PATCOUNT	8
NURCOUNT	4
DOCCOUNT	2

PATIENT segment

The following table shows the PATIENT segment, which has two fields:

- The patient number (PATNUM)
- The patient name (PATNAME)

PATNUM is a unique key field.

Field name	Field length (in bytes)
PATNUM	12
PATNAME	17

ILLNESS segment

The following table shows the ILLNESS segment, which has one field:

- The illness name (ILLNAME)

Field name	Field length (in bytes)
ILLNAME	15

TREATMNT segment

The following table shows the TREATMNT segment, which has three fields:

- The day of treatment (TREATDAY)
- The type of treatment (TREATMNT)
- The treatment comments (COMMENTS)

Field name	Field length (in bytes)
TREATDAY	8
TREATMNT	15
COMMENTS	10

DOCTOR segment

The following table shows the DOCTOR segment, which has two fields:

- The doctor number (DOCTNO)
- The doctor name (DOCNAME)

Field name	Field length (in bytes)
DOCTNO	4
DOCNAME	20

BILLING segment

The following table shows the BILLING segment, which has two fields:

- The bill amount (AMOUNT)
- The bill comments (COMMENTS)

Field name	Field length (in bytes)
AMOUNT	8
COMMENTS	20

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Programming using the IMS Universal Database resource adapter

This information describes how to write programs with the IMS Universal Database resource adapter to access IMS databases.

Overview of the IMS Universal Database resource adapter

The IMS Universal Database resource adapter is based on the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) 1.5 standard. The purpose of the JCA is to connect Enterprise Information Systems (EISs), such as IMS, into the Java EE platform. JCA provides a number of services that are managed by a Java EE application server. These services include security credential management, connection pooling, and transaction management.

These services are provided by means of system level contracts between the IMS Universal Database resource adapter and the Java EE application server, without the need for additional coding by the application programmer.

The JCA specification defines a programming interface called the Common Client Interface (CCI). This interface is used to communicate with any EIS. The IMS Universal Database resource adapter implements the CCI for interactions with IMS databases. The CCI interfaces for the IMS Universal Database resource adapter are in the com.ibm.ims.db.cci package. The CCI implementation provided by IMS allows applications to make either SQL or DL/I calls to access the IMS database.

In addition to the CCI interface provided by the IMS Universal Database resource adapter, you can also write JDBC applications to access your IMS data from a managed environment, while leveraging the Java EE services provided by the application server. This capability is provided by the IMS Universal JCA/JDBC driver version of the IMS Universal Database resource adapter. The IMS Universal JCA/JDBC driver is based on the Java Platform, Enterprise Edition (Java EE) Connector Architecture (JCA) 1.5 and Java Database Connectivity (JDBC) 3.0 standard.

The IMS Universal Database resource adapter communicates with IMS Connect as the TCP/IP endpoint to access IMS.

Preparing to write a Java application with the IMS Universal drivers

Java application programs that use the IMS Universal drivers require the Java Development Kit 5.0 (JDK 5.0). Java programs that run in JMP and JBP regions require the Java Development Kit 6.0 (JDK 6.0) or later. Java application programs that use the IMS Universal drivers also require a way to generate the IMS database metadata, such as using the IMS Enterprise Suite Explorer for Development. The default segment encoding of the database metadata class produced by the IMS Explorer for Development is `cp1047`. To change the segment encoding, use the com.ibm.ims.base.DLIBaseSegment.setDefaultEncoding method.

Transaction types and programming interfaces supported by the IMS Universal Database resource adapter

The IMS Universal Database resource adapter provides four types of support for optimized transaction management and performance.

The types of transaction support provided by the IMS Universal Database resource adapter are:

IMS Universal Database resource adapter with local transaction support (imsudbLocal.rar)
This resource adapter provides a CCI programming interface and LocalTransaction support when deployed on any supported Java EE application server.

IMS Universal Database resource adapter with XA transaction support (imsudbXA.rar)
This resource adapter provides a CCI programming interface and both XATransaction and LocalTransaction support when deployed on any supported Java EE application server.

IMS Universal JCA/JDBC driver with local transaction support (imsudbJLocal.rar)
This resource adapter provides a JDBC programming interface and LocalTransaction support when deployed on any supported Java EE application server.

IMS Universal JCA/JDBC driver with XA transaction support (imsudbJXA.rar)
This resource adapter provides a JDBC programming interface and both XATransaction and LocalTransaction support when deployed on any supported Java EE application server.

Restriction: XA transaction support is available only with type-4 connectivity.

For global or two-phase commit transaction processing, use the IMS Universal Database resource adapters with XA transaction support. For single-phase commit functionality, use either the IMS Universal Database resource adapters with XA transaction support or with local transaction support.

In order to provide for different transactional qualities of service for Java EE applications, it is possible to deploy two or more separate types of IMS Universal Database resource adapters into the same Java EE application server.

When carrying out multiple interactions with IMS databases using the IMS Universal Database resource adapter, you might want to group all actions together to ensure that they either all succeed or all fail. This can be done using container-managed or bean-managed transaction demarcation.

In *container-managed transactions*, all work performed in an EJB method invocation is part of one unit of work, and no explicit demarcation by the application is required. Transactional integrity is managed by the Java EE application server.

In *bean-managed transactions*, you must use the javax.resource.cci.LocalTransaction or javax.transaction.UserTransaction interface to programmatically demarcate units of work explicitly. Bean-managed transactions that use the LocalTransaction interface can group work performed only through the resource adapter; the UserTransaction interface allows all transactional resources within the application to be grouped. Use a bean-managed EJB if you need to have multiple units of works within the same EJB method invocation.

When using the type-2 IMS Universal Database resource adapter, if you specify a driverType connection property of 2, you can use javax.resource.cci.LocalTransaction for bean-managed transactions or the JDBC Connection interface. If you specify a driverType connection property of 2_CTX, you can use the javax.transaction.UserTransaction for application programs that issue explicit commit and rollback calls.

Connecting to IMS with the IMS Universal Database resource adapter

The IMS Universal Database resource adapter provides connectivity to IMS databases from a Java EE-managed environment.

Important: Support for type-2 connectivity, including the RRSLocalOption, was added to the IMS Universal drivers by APARs PM14766 and PK99686.

The Common Client Interface (CCI) provides ConnectionFactory and Connection interfaces to establish a connection with an Enterprise Information System (EIS). When using the CCI programming interface with the IMS Universal Database resource adapter, your Java application component looks up a ConnectionFactory instance using the Java Naming and Directory Interface (JNDI) and uses the ConnectionFactory instance to get a connection to an IMS database.

Similarly, when using the JDBC programming interface with the IMS Universal JCA/JDBC driver, your Java application component looks up a DataSource instance using JNDI, and uses the DataSource instance to obtain a Connection object

RRSLocalOption connectivity type

In addition to type-4 and type-2 connectivity, the RRSLocalOption connectivity type is supported by the IMS Universal Database resource adapter running on WebSphere Application Server for z/OS. With RRSLocalOption connectivity, applications using the IMS Universal Database resource adapter do not issue commit or rollback calls. Instead, transaction processing is managed by WebSphere Application Server for z/OS. Two-phase commit (XA) transaction processing is not supported with RRSLocalOption connectivity type.

Connecting using the IMS Universal Database resource adapter in a managed environment

In a managed (or three-tier) environment, your Java EE application interacts with a Java EE application server, such as WebSphere Application Server, and the IMS Universal Database resource adapter to communicate with an IMS database.

Important: Support for type-2 connectivity, including the RRSLocalOption, was added to the IMS Universal drivers by APARs PM14766 and PK99686.

To configure and use a CCI Connection object in WebSphere Application Server to access an IMS database:

1. Deploy the IMS Universal Database resource adapter in WebSphere Application Server using the administrative console.
2. Create a connection factory for use with the IMS Universal Database resource adapter in WebSphere Application Server through the administrative console.
   a. Specify a name for the connection factory and a Java Naming and Directory Interface (JNDI) name.
   b. Set the following custom connection properties for the IMS Universal Database resource adapter:

      DatastoreName

      The name of the IMS data store to access.

- When using type-4 connectivity, the **DatastoreName** property must match either the name of the data store defined to ODBM or be blank. The data store name is defined in the ODBM CSLDCxxx PROCLIB member using either the DATASTORE(NAME=*name*) or DATASTORE(NAME=*name*, ALIAS(NAME=*aliasname*)) parameter. If an alias is specified, you must specify the *aliasname* as the value of the **datastoreName** property. If the **DatastoreName** value is left blank (or not supplied), IMS Connect connects to any available instance of ODBM as it is assumed that data sharing is enabled between all datastores defined to ODBM.
- When using type-2 connectivity, set the **DatastoreName** property to the IMS subsystem alias. This is not required to be set for the Java Dependent Region run time.

MetadataURL

The location of the database metadata representing the target IMS database.

The **MetadataURL** property is the fully qualified name of the Java metadata class generated by the IMS Enterprise Suite Explorer for Development. The URL must be prefixed with `class://` (for example, `class://com.foo.BMP255DatabaseView`).

In a J2C Connection Factory environment, the **MetadataURL** property can be overridden for an individual connection without affecting the default value specified for the resource adapter.

PortNumber

The TCP/IP server port number to be used to communicate with IMS Connect. The port number is defined using the DRDAPORT parameter on the ODACCESS statement in the IMS Connect configuration PROCLIB member. The default port number is 8888. Do not set this property when using type-2 connectivity.

DatastoreServer

The name or IP address of the data store server (IMS Connect). You can provide either the host name (for example, dev123.svl.ibm.com) or the IP address (for example, 192.166.0.2). Do not set this property when using type-2 connectivity.

DriverType

The type of driver connectivity to use. The **DriverType** value must be "4" for type-4 connectivity or "2" for type-2 connectivity. If the driver is running on WebSphere Application Server for z/OS, you can also set the **DriverType** value to "2_CTX" for RRSLocalOption connectivity.

user The user name for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

password

The password for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

currentSchema

Optional. Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements.

dbViewLocation
Optional. Specifies the fully qualified path to a databaseView metadata class. You can use this property to include a metadata class that is not located in your project path.

dpsbOnCommit
Optional. Set this property to **true** to deallocate the PSB when a commit occurs.

Recommendation: Do not set this property to **true** except in a managed environment with integrated connection pooling.

fetchSize
Optional. Gives the client a hint about the number of rows to get from the database when more rows are needed. The number specified for this property only affects data retrieved with the current connection. If the value specified is 0, all of the applicable rows are returned.

The default value for this property is 1 for unmanaged connections and 0 for managed connections.

llField Optional. Setting this property to **true** exposes the LL field data as a normal column in the result set. You can modify the LL field value to change the length of a variable length segment instance.

maxRows
Optional. Specifies the maximum number of rows to return in a query result set. The default value is 0, which returns all of the applicable rows in the result set.

traceFile
Optional. Specifies the name of the trace file for the connection.

traceFileAppend
Optional. If the specified trace file exists, setting this property to true specifies that the trace data for the new connection must be appended to the existing trace file instead of overwriting it.

This property is ignored if no value is specified for **traceFile**.

traceDirectory
Optional. Specifies the file system directory where the trace file is located. By default, this path is the directory where the application is executed.

This property is ignored if no value is specified for **traceFile**.

traceLevel
Optional. Specifies which traces are enabled for the connection. The valid values for this property are defined in the Java API documentation for the IMSDataSource class.

By default, all traces are disabled.

This property is ignored if no value is specified for **traceFile**.

3. In the deployment descriptor for your Java EE application, add a resource reference for the connection factory that was created in the previous step. Set the name of the resource reference to the JNDI name of the connection factory and set the type to javax.resource.cci.ConnectionFactory.
4. In your Java EE application, create an initial JNDI naming context and get the corresponding javax.resource.cci.ConnectionFactory instance for the IMS

Universal Database resource adapter using JNDI lookup. The following code sample shows how to perform the JNDI lookup to obtain a ConnectionFactory instance, where the connection factory has the JNDI name “imsdblocal”:

```
Context initctx = new InitialContext();
javax.resource.cci.ConnectionFactory cf =
 (javax.resource.cci.ConnectionFactory)initctx.lookup
 ("java:comp/env/imsdblocal");
```

5. Create a com.ibm.ims.db.cci.IMSConnectionSpec object and, if necessary, set application-specific property values to override the values are already assigned in the connection factory deployment descriptor for accessing the IMS database. The following code sample shows how to create the IMSConnectionSpec object, assuming the application needs to override the values for the user ID and password:

   ```
   IMSConnectionSpec connSpec = new IMSConnectionSpec();
   connSpec.setUser("myUserId");
   connSpec.setPassword("myPassword");
   ```

6. Get the connection to IMS from the connection factory by invoking the getConnection method and passing in the IMSConnectionSpec instance created in the previous step. If the application does not need to override any of the connection properties set in the connection factory deployment descriptor, then there is no need to instantiate a IMSConnectionSpec object. Instead, the application can invoke the getConnection method that takes no arguments. The returned javax.resource.cci.Connection instance represents an application-level handle to the underlying physical connection.
7. Use the connection to access the IMS database using the CCI Interaction interface. The following code sample shows how to obtain the Connection object:

   ```
   javax.resource.cci.Connection conn = cf.getConnection(connSpec);
   ```

8. After your Java EE application is finished with the connection, close the connection with the close method on the Connection interface.

Example code for connecting to an IMS database using the IMS Universal Database resource adapter in a managed environment

The following code sample shows the flow for connecting to an IMS database using the IMS Universal Database resource adapter from an Java EE application:

```
//obtain the initial JNDI Naming context
Context initctx = new InitialContext();

//perform JNDI lookup to obtain the connection factory
javax.resource.cci.ConnectionFactory cf =
   (javax.resource.cci.ConnectionFactory)initctx.lookup("java:comp/env/imsdblocal");

//specify connection properties
IMSConnectionSpec connSpec = new IMSConnectionSpec();
connSpec.setUser("user");
connSpec.setPassword("password");

//create CCI connection
javax.resource.cci.Connection conn = cf.getConnection(connSpec);
```

The following code sample shows how you can override the default **MetadataURL** value of the resource adapter for a specific connection. Overriding the value in this way does not alter the default value or require any modifications to the J2C connection factory parameters.

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("myJNDIName");
Connection con = ((IMSHybridDataSource)ds).getConnection(iSpec);
```

Related tasks:

"Configuring the IMS Universal drivers for SSL support" on page 637

Related reference:

IMS Universal drivers support for the Common Client Interface (Application Programming APIs)

Connecting using the IMS Universal JCA/JDBC driver in a managed environment

To access IMS databases using a JDBC programming interface in a managed (or three-tier) environment, you need to deploy the IMS Universal JCA/JDBC driver on your Java EE application server and configure the connection properties.

Important: Support for type-2 connectivity, including the RRSLocalOption, was added to the IMS Universal drivers by APARs PM14766 and PK99686.

To configure and use the IMS Universal JCA/JDBC driver to access an IMS database:

1. Deploy the IMS Universal JCA/JDBC driver in WebSphere Application Server using the administrative console.
2. Create a connection factory for use with the IMS Universal JCA/JDBC driver in WebSphere Application Server through the administrative console.
   a. Specify a name for the connection factory and a Java Naming and Directory Interface (JNDI) name. Set the connection factory interface as javax.sql.DataSource.
   b. Set the following custom connection properties for the IMS Universal JCA/JDBC driver connection factory:

      DatastoreName

      The name of the IMS data store to access.

      - When using type-4 connectivity, the **DatastoreName** property must match either the name of the data store defined to ODBM or be blank. The data store name is defined in the ODBM CSLDCxxx PROCLIB member using either the DATASTORE(NAME=*name*) or DATASTORE(NAME=*name*, ALIAS(NAME=*aliasname*)) parameter. If an alias is specified, you must specify the *aliasname* as the value of the **datastoreName** property. If the **DatastoreName** value is left blank (or not supplied), IMS Connect connects to any available instance of ODBM as it is assumed that data sharing is enabled between all datastores defined to ODBM.
      - When using type-2 connectivity, set the **DatastoreName** property to the IMS subsystem alias. This is not required to be set for the Java Dependent Region run time.

      MetadataURL

      The location of the database metadata representing the target IMS database.

      The **MetadataURL** property is the fully qualified name of the Java metadata class generated by the IMS Enterprise Suite Explorer for Development. The URL must be prefixed with `class://` (for example, `class://com.foo.BMP255DatabaseView`).

In a J2C Connection Factory environment, the **MetadataURL** property can be overridden for an individual connection without affecting the default value specified for the resource adapter.

PortNumber
The TCP/IP server port number to be used to communicate with IMS Connect. The port number is defined using the DRDAPORT parameter on the ODACCESS statement in the IMS Connect configuration PROCLIB member. The default port number is 8888. Do not set this property when using type-2 connectivity.

DatastoreServer
The name or IP address of the data store server (IMS Connect). You can provide either the host name (for example, dev123.svl.ibm.com) or the IP address (for example, 192.166.0.2). Do not set this property when using type-2 connectivity.

DriverType
The type of driver connectivity to use. The **DriverType** value must be "4" for type-4 connectivity or "2" for type-2 connectivity. If the driver is running on WebSphere Application Server for z/OS, you can also set the **DriverType** value to "2_CTX" for RRSLocalOption connectivity.

sslConnection
Optional. Indicates if this connection uses Secure Sockets Layer (SSL) for data encryption. Set this property to "true" to enable SSL, or to "false" otherwise. Do not set this property when using type-2 connectivity.

loginTimeout
Optional. Specifies the number of seconds that the driver waits for a response from the server before timing out a connection initialization or server request. Set this property to a non-negative integer for the number of seconds. Set this property to 0 for an infinite timeout length. Do not set this property when using type-2 connectivity.

user The user name for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

password
The password for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

currentSchema
Optional. Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements.

dbViewLocation
Optional. Specifies the fully qualified path to a databaseView metadata class. You can use this property to include a metadata class that is not located in your project path.

dpsbOnCommit
Optional. Set this property to **true** to deallocate the PSB when a commit occurs.

Recommendation: Do not set this property to **true** except in a managed environment with integrated connection pooling.

fetchSize
Optional. Gives the client a hint about the number of rows to get from the database when more rows are needed. The number specified for this property only affects data retrieved with the current connection. If the value specified is 0, all of the applicable rows are returned.

The default value for this property is 1 for unmanaged connections and 0 for managed connections.

llField Optional. Setting this property to **true** exposes the LL field data as a normal column in the result set. You can modify the LL field value to change the length of a variable length segment instance.

maxRows
Optional. Specifies the maximum number of rows to return in a query result set. The default value is 0, which returns all of the applicable rows in the result set.

traceFile
Optional. Specifies the name of the trace file for the connection.

traceFileAppend
Optional. If the specified trace file exists, setting this property to true specifies that the trace data for the new connection must be appended to the existing trace file instead of overwriting it.

This property is ignored if no value is specified for **traceFile**.

traceDirectory
Optional. Specifies the file system directory where the trace file is located. By default, this path is the directory where the application is executed.

This property is ignored if no value is specified for **traceFile**.

traceLevel
Optional. Specifies which traces are enabled for the connection. The valid values for this property are defined in the Java API documentation for the IMSDataSource class.

By default, all traces are disabled.

This property is ignored if no value is specified for **traceFile**.

3. In the deployment descriptor for your Java EE application, add a resource reference for the connection factory that was created in the previous step. Set the name of the resource reference to the JNDI name of the connection factory and set the type to javax.resource.cci.ConnectionFactory.
4. In your Java EE application, create an initial JNDI naming context and get a javax.sql.DataSource instance for the IMS Universal JCA/JDBC hybrid driver using JNDI lookup. The following code sample shows how to perform the JNDI lookup to obtain a DataSource instance, where the connection factory has the JNDI name “imsdblocal”:

```
InitialContext ic = new InitialContext();
javax.sql.DataSource ds =
 (DataSource)ic.lookup("java:comp/env/imsdblocal");
```

5. Use the getConnection method on the DataSource instance to obtain a java.sql.Connection instance, as shown by the following code sample:

```
Connection con = ds.getConnection();
```

6. After your Java EE application has finished with the connection, close the connection using the close method on the Connection interface.

Example code for connecting to an IMS database using the IMS Universal JCA/JDBC driver

The following code sample shows the flow for connecting to an IMS database using the IMS Universal JCA/JDBC driver from an Java EE application:

```
//obtain the initial JNDI Naming context
InitialContext ic = new InitialContext();

//perform JNDI lookup to obtain the data source
javax.sql.DataSource ds =
 (DataSource)ic.lookup("java:comp/env/imsdblocal");

//specify connection properties
ds.setUser("myUserID");
ds.setPassword("myPassword");
props.put("sslConnection", "true");
props.put("loginTimeout", "10");

//create JDBC connection
java.sql.Connection con = ds.getConnection();
```

Related tasks:

"Configuring the IMS Universal drivers for SSL support" on page 637

Related reference:

javax.sql.DataSource methods supported (Application Programming APIs)

Sample EJB application using the IMS Universal Database resource adapter CCI programming interface

The following sample EJB bean demonstrates the basic programming flow for a JCA application using the IMS Universal Database resource adapter in a managed environment.

```
package client;

import java.sql.SQLException;
import javax.naming.InitialContext;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Interaction;
import javax.resource.cci.ResultSet;
import javax.transaction.UserTransaction;
import com.ibm.ims.db.cci.SQLInteractionSpec;

/**
 * Bean implementation class for Enterprise Bean: StatefulBeanManaged
 */
public class BeanManagedSampleBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext mySessionCtx;

    public void execute() throws Exception {
        InitialContext ic = new InitialContext();
        ConnectionFactory cf =
     (ConnectionFactory) ic.lookup("java:comp/env/MyMCF");
        Connection conn = null;
        UserTransaction ut = null;
```

```
            try {
                ut = this.mySessionCtx.getUserTransaction();
                ut.begin();

                conn = cf.getConnection();
                Interaction ix = conn.createInteraction();
                SQLInteractionSpec iSpec = new SQLInteractionSpec();

                // This query will return information for each person
                // in the phonebook with the last name WATSON
                iSpec.setSQL("SELECT * FROM " +
          "PCB01.PHONEBOOK WHERE LASTNAME='WATSON'");

                ResultSet rs = (ResultSet) ix.execute(iSpec, null);

                // Print out the first name of every person in the
                // phonebook with the last name WATSON
                while (rs.next()) {
                    System.out.println(rs.getString("FIRSTNAME"));
                }

                rs.close();
                ix.close();
                ut.commit();
                conn.close();
            } catch (ResourceException e) {
                ut.rollback();
                conn.close();
            } catch (SQLException e) {
                ut.rollback();
                conn.close();
            }
        }

        /**
         * getSessionContext
         */
        public javax.ejb.SessionContext getSessionContext() {
            return mySessionCtx;
        }

        /**
         * setSessionContext
         */
        public void setSessionContext(javax.ejb.SessionContext ctx) {
            mySessionCtx = ctx;
        }

        /**
         * ejbCreate
         */
        public void ejbCreate() throws javax.ejb.CreateException {
        }

        /**
         * ejbActivate
         */
        public void ejbActivate() {
        }

        /**
         * ejbPassivate
         */
        public void ejbPassivate() {
        }

        /**
```

```
     * ejbRemove
     */
    public void ejbRemove() {
    }
}
```

Accessing IMS data with the DLIInteractionSpec class

Use the DLIInteractionSpec class to retrieve, insert, update, and delete data from an IMS database using DL/I-like programming semantics with the IMS Universal Database resource adapter.

Before your application component can retrieve, insert, update, or delete data from an IMS database, you need to obtain a javax.resource.cci.Connection instance for the physical connection to the database.

To retrieve, insert, update, or delete data using the DLIInteractionSpec class:

1. In your application component, create a new javax.resource.cci.Interaction instance using the Connection.createInteraction method. For example, in the following code sample, *con* is a javax.resource.cci.Connection instance for an IMS database:

   ```
   Interaction ix = con.createInteraction();
   ```

2. Create a new com.ibm.ims.db.cci.DLIInteractionSpec instance.

   ```
   DLIInteractionSpec iSpec = new DLIInteractionSpec();
   ```

3. Set the function to perform using the DLIInteractionSpec.setFunctionName method, and specifying the function constant value listed in the table below as the input parameter.

Data operation to perform	setFunctionName value
Data retrieval	DLIInteractionSpec.RETRIEVE
Data insertion	DLIInteractionSpec.CREATE
Data update	DLIInteractionSpec.UPDATE
Data deletion	DLIInteractionSpec.DELETE

   For example, the following code sample specifies a data retrieval operation:

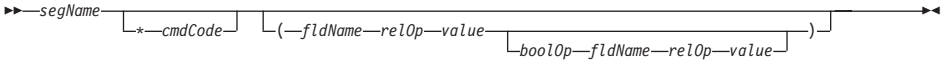
   ```
   iSpec.setFunctionName(DLIInteractionSpec.RETRIEVE);
   ```

4. Set the PCB name using the DLIInteractionSpec.setPCBName method. For example, the following code sample specifies the PCB to be used for this interaction as “PCB01”:

   ```
   iSpec.setPCBName("PCB01");
   ```

5. Set the segment search argument (SSA) list using the DLIInteractionSpec.setSSAList method. The setSSAList method allows you to specify an SSA in a syntax similar to traditional DL/I.
   - You can manually provide the SSA qualification statement as a string in the argument. The syntax is as follows:

     Syntax for segment search argument qualification statement in the setSSAList method



segName
The name of the segment as defined in the Java metadata class generated by the IMS Enterprise Suite Explorer for Development.

cmdCode (optional)
All DL/I command codes except Q and C are supported.

fldName
The name of the field.

relOp The SSA qualification statement's relational operator. Supported values are:

= Equals

!= Not equal

> Greater than

>= Greater than or equals

< Less than

<= Less than or equals

value A string representation of the field value. If the value is character-based, the string has to be enclosed in quotation marks. If the value is numeric, it does not need to be enclosed in quotation marks. If the character-based value has quotation marks, use a single quote as an escape for the quote in the value. For example, if the value is *"O'brian"*, you would enter it as *"O''brian"*.

boolOp
Boolean operators for adding additional field-level qualifications. The supported Boolean operators are:
- logical AND (specified * or &)
- logical OR (specified + or |)
- independent AND (specified #)

The following code example shows how to set the segment search argument list to return the last patient admitted to all wards with more than five doctors and less than three nurses in hospital "ALEXANDRIA". The *L command means "last occurrence".

```
String ssaList =
"Hospital(HospName='ALEXANDRIA') Ward(Doccount>5 | Nurcount<3) Patient *L";
iSpec.setSSAList(ssaList);
```

- Instead of providing the string manually, you can use the com.ibm.ims.db.cci.SSAListHelper class to generate the string.

  The following code example shows how to set the SSA qualification statement string using the SSAListHelper:

```
SSAListHelper sh = new SSAListHelper();
sh.addInitialQualification
  ("Hospital","HospName",SSAListHelper.EQUALS, "ALEXANDRIA");
sh.addInitialQualification("Ward","Doccount",
   SSAListHelper.GREATER_THAN, 5);
sh.appendQualification("Ward",SSAListHelper.OR, "Nurcount",
   SSAListHelper.LESS_THAN, 3);
sh.addCommandCode("Patient",SSAListHelper.CC_L);
iSpec.setSSA(sh.toString());
```

6. Create a javax.resource.cci.RecordFactory instance using the ConnectionFactory.getRecordFactory method. For example, the following code sample creates a ConnectionFactory instance *rf*:

```
RecordFactory rf = cf.getRecordFactory();
```

This step is not needed for a DELETE operation.

7. Create a javax.resource.cci.MappedRecord instance using the RecordFactory.getMappedRecord method. For a RETRIEVE operation, pass the name of the record you want to create as an argument to this method. For a CREATE or UPDATE operation, the argument is the name of the segment to insert or update. For example, in the following code sample for a RETRIEVE operation, *rf* is a javax.resource.cci.RecordFactory instance:

   ```
   MappedRecord input = rf.createMappedRecord("myHospitalRecord");
   ```

   This step is not needed for a DELETE operation because the MappedRecord is not used.

8. Specify the field to target for the data operation using the MappedRecord.put method. Pass the name of the field as the first argument to this method. For a CREATE or UPDATE operation, the MappedRecord is used to specify the field to insert or update as well as its values. Pass in the value of the field as the second argument. For a RETRIEVE operation, the MappedRecord is used to specify the field that the application is interested in retrieving as a result of the call, and the second argument in the put method call is ignored (you can pass in a null). If you do not specify any fields, the RETRIEVE operation will return all the fields of the leaf segment for that record, along with all the fields in segments which have SSAs specified with a *D command. For CREATE, UPDATE, and RETRIEVE operations, you can specify multiple fields by making multiple MappedRecord.put method calls. For example, in the following code sample, *input* is a javax.resource.cci.MappedRecord instance and "HospCode" is the name of the field we want to retrieve:

   ```
   input.put("HospCode", null);
   ```

   This step is not needed for a DELETE operation as the MappedRecord is not used.

9. Execute the query by calling the Interaction.execute method. Pass the DLIInteractionSpec object and the MappedRecord object as arguments. If the query is successful, the method returns a Record object with the query results. You can cast the Record instance to javax.resource.cci.ResultSet and process the results as tabular data in your application component. For example, in the following code sample, *results* is a javax.resource.cci.ResultSet instance, *ix* is a javax.resource.cci.Interaction instance, *iSpec* is a com.ibm.ims.db.cci.DLIInteractionSpec instance, and *input* is a javax.resource.cci.MappedRecord instance:

   ```
   results = (ResultSet)ix.execute(iSpec, input);
   ```

Example code for IMS data operations using the DLIInteraction interface

The following complete code example shows how to use the DLIInteraction interface to retrieve fields from a WARD segment.

```
package client;

import java.sql.SQLException;
import javax.naming.InitialContext;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Interaction;
import javax.resource.cci.ResultSet;
```

```
import javax.transaction.UserTransaction;
import com.ibm.ims.db.cci.SQLInteractionSpec;

/**
 * Bean implementation class for Enterprise Bean: StatefulBeanManaged
 */
public class BeanManagedSampleDLIBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext mySessionCtx;

    public void execute() throws Exception {
        InitialContext ic = new InitialContext();
        ConnectionFactory cf =
    (ConnectionFactory) ic.lookup("java:comp/env/MyMCF");
        Connection conn = null;
        UserTransaction ut = null;

        try {
            ut = this.mySessionCtx.getUserTransaction();
            ut.begin();

            conn = cf.getConnection();
            Interaction ix = conn.createInteraction();

            DLIInteractionSpec iSpec = new DLIInteractionSpec();
            iSpec.setFunctionName("RETRIEVE");
            iSpec.setPCBName("PCB09");

            // This query will return the WARDNAME, PATCOUNT, DOCCOUNT,
            // and NURCOUNT fields for all WARDs with WARNNO = 51
            iSpec.setSSAList("WARD (WARDNO = '51')");

            // Create RecordFactory
            RecordFactory rf = cf.getRecordFactory();

            // Create Record
            MappedRecord input = rf.createMappedRecord("WARD");
            // Specify the fields to retrieve
            input.put("WARDNAME", null);
            input.put("PATCOUNT", null);
            input.put("DOCCOUNT", null);
            input.put("NURCOUNT", null);

            ResultSet results = (ResultSet) ix.execute(iSpec, input);
            while (results.next()) {
                System.out.println(results.getString("WARDNAME"));
                System.out.println(results.getString("PATCOUNT"));
                System.out.println(results.getString("DOCCOUNT"));
                System.out.println(results.getString("NURCOUNT"));
            }

            rs.close();
            ix.close();
            ut.commit();
            conn.close();
        } catch (ResourceException e) {
            ut.rollback();
            conn.close();
        } catch (SQLException e) {
            ut.rollback();
            conn.close();
        }
    }

    /**
     * getSessionContext
     */
```

```
    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }

    /**
     * setSessionContext
     */
    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }

    /**
     * ejbCreate
     */
    public void ejbCreate() throws javax.ejb.CreateException {
    }

    /**
     * ejbActivate
     */
    public void ejbActivate() {
    }

    /**
     * ejbPassivate
     */
    public void ejbPassivate() {
    }

    /**
     * ejbRemove
     */
    public void ejbRemove() {
    }
}
```

Related concepts:

"Connecting to IMS with the IMS Universal Database resource adapter" on page 563

Related reference:

IMS Universal drivers support for the Common Client Interface (Application Programming APIs)

Accessing IMS data with the SQLInteractionSpec class

Use the SQLInteractionSpec class to retrieve, insert, update, and delete data from an IMS database using SQL queries with the IMS Universal Database resource adapter. The IMS Universal Database resource adapter supports the same SQL statement syntax and usage as the IMS Universal JDBC driver and has the same restrictions.

Before your application component can retrieve, insert, update, or delete data from an IMS database, you need to obtain a javax.resource.cci.Connection instance for the physical connection to the database.

To retrieve, insert, update, or delete data using the SQLInteractionSpec class:

1. In your application component, create a new javax.resource.cci.Interaction instance using the Connection.createInteraction method. For example, in the following code sample, *con* is a javax.resource.cci.Connection instance for an IMS database:

   ```
   Interaction ix = con.createInteraction();
   ```

2. Create a new com.ibm.ims.db.cci.SQLInteractionSpec instance.

   ```
   SQLInteractionSpec iSpec = new SQLInteractionSpec();
   ```

3. Set the SQL query string using the SQLInteractionSpec.setSQL method. In the query, you can specify the qualification column values in the WHERE clause with a ? parameter marker, meaning that the values will provided later (similar to a PreparedStatement in JDBC).

   - The following example shows how to specify the SELECT statement without using parameter markers, where *iSpec* is an instance of SQLInteractionSpec:

     ```
     iSpec.setQuery("SELECT PATIENT.PATNAME, ILLNESS.ILLNAME "+
          "FROM pcb01.HOSPITAL, pcb01.PATIENT, pcb01.ILLNESS " +
          "WHERE HOSPITAL.HOSPNAME='SANTA TERESA'");
     ```

   - The following example shows how to perform the SELECT statement with parameter markers, where *iSpec* is an instance of SQLInteractionSpec:

     ```
     iSpec.setQuery("SELECT PATIENT.PatName, WARD.WardName "+
          "FROM pcb01.HOSPITAL, pcb01.PATIENT, pcb01.WARD " +
          "WHERE HOSPITAL.HospName=? AND WARD.DocCount>?");
     ```

4. Create a javax.resource.cci.RecordFactory instance using the ConnectionFactory.getRecordFactory method. Creating a RecordFactory is not needed for SQL queries without parameter markers.

5. Create a javax.resource.cci.IndexedRecord instance using the RecordFactory.getIndexedRecord method. Pass the name of the record you want to create as an argument to this method. For example, in the following code sample, *rf* is a javax.resource.cci.RecordFactory instance:

   ```
   IndexedRecord input = rf.createIndexedRecord("myPatientRecord");
   ```

   Creating a IndexedRecord is not needed for SQL queries without parameter markers.

6. If your query string uses parameter markers, use the IndexedRecord.add method to qualify the WHERE clause. For example, using the same query string with parameter markers as in step 5, where *input* is an instance of IndexedRecord:

   ```
   input.add(1, "Santa Teresa"); //HospName value is "Santa Teresa"
   input.add(2, 5);              //DocCount value is greater than 5
   ```

7. Execute the query by calling the Interaction.execute method. Pass the SQLInteractionSpec object and the IndexedRecord object as arguments. If your SQL query does not use parameter markers, the second argument in the execute method call is ignored (you can pass in a null). If the query is successful, the method returns a Record object with the query results. You can cast the Record instance to javax.resource.cci.ResultSet and process the results as tabular data in your application component. For example, in the following code sample, *results* is a javax.resource.cci.ResultSet instance, *ix* is a javax.resource.cci.Interaction instance, *iSpec* is a com.ibm.ims.db.cci.SQLInteractionSpec instance, and *input* is a javax.resource.cci.IndexedRecord instance:

   ```
   results = (ResultSet)ix.execute(iSpec, input);
   ```

Example code for IMS data operations using the SQLInteractionSpec class

The following code example shows how to use the SQLInteractionSpec class to retrieve patient names from PATIENT records.

```
package client;

import java.sql.SQLException;
```

```
import javax.naming.InitialContext;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Interaction;
import javax.resource.cci.ResultSet;
import javax.transaction.UserTransaction;
import com.ibm.ims.db.cci.SQLInteractionSpec;

/**
 * Bean implementation class for Enterprise Bean: StatefulBeanManaged
 */
public class BeanManagedSampleSQLBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext mySessionCtx;

    public void execute() throws Exception {
        InitialContext ic = new InitialContext();
        ConnectionFactory cf =
     (ConnectionFactory) ic.lookup("java:comp/env/MyMCF");
        Connection conn = null;
        UserTransaction ut = null;

        try {
            ut = this.mySessionCtx.getUserTransaction();
            ut.begin();

            conn = cf.getConnection();
            Interaction ix = conn.createInteraction();
            SQLInteractionSpec iSpec = new SQLInteractionSpec();

            // This query will return the WARDNAME, PATCOUNT, DOCCOUNT,
            // and NURCOUNT fields for the WARD with WARDNO = 51
            iSpec.setSQL("SELECT WARDNAME, PATCOUNT, DOCCOUNT, " +
      "NURCOUNT FROM PCB09.WARD WHERE WARDNO='51'");

            ResultSet rs = (ResultSet) ix.execute(iSpec, null);

            while (rs.next()) {
                System.out.println(rs.getString("WARDNAME"));
                System.out.println(rs.getString("PATCOUNT"));
                System.out.println(rs.getString("DOCCOUNT"));
                System.out.println(rs.getString("NURCOUNT"));
            }

            rs.close();
            ix.close();
            ut.commit();
            conn.close();
        } catch (ResourceException e) {
            ut.rollback();
            conn.close();
        } catch (SQLException e) {
            ut.rollback();
            conn.close();
        }
    }

    /**
     * getSessionContext
     */
    public javax.ejb.SessionContext getSessionContext() {
        return mySessionCtx;
    }

    /**
     * setSessionContext
```

```
     */
    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }

    /**
     * ejbCreate
     */
    public void ejbCreate() throws javax.ejb.CreateException {
    }

    /**
     * ejbActivate
     */
    public void ejbActivate() {
    }

    /**
     * ejbPassivate
     */
    public void ejbPassivate() {
    }

    /**
     * ejbRemove
     */
    public void ejbRemove() {
    }
}
```

Related concepts:

"Connecting to IMS with the IMS Universal Database resource adapter" on page 563

Related reference:

IMS Universal drivers support for the Common Client Interface (Application Programming APIs)

"SQL statement usage with the IMS Universal JDBC driver" on page 596

Accessing IMS data with the IMS Universal JCA/JDBC driver

Use the IMS Universal JCA/JDBC driver if you require full use of the IMS Universal JDBC driver within a Java EE runtime environment.

Before your Java EE application component can retrieve, insert, update, or delete data from an IMS database, you need to obtain a java.sql.Connection instance for the physical connection to the database.

In your Java EE application component, code the application logic for the data operations you want to perform in the same way as for a JDBC application. The IMS Universal JCA/JDBC driver has the same SQL statement syntax support and usage restrictions as the IMS Universal JDBC driver.

Example EJB application using the IMS Universal JCA/JDBC driver

The following code sample shows a bean-managed EJB application that connects to an IMS database, retrieves a list of patient names using a SQL SELECT query, and modifies the patient information using a SQL UPDATE query.

```
package client;
import java.sql.SQLException;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

import javax.transaction.UserTransaction;
import com.ibm.ims.db.cci.SQLInteractionSpec;

/**
 * Bean implementation class for Enterprise Bean: StatefulBeanManaged
 */
public class JDBCBeanManagedSampleSQLBean {

 private javax.ejb.SessionContext mySessionCtx;

 public void execute() throws Exception {
  InitialContext ic = new InitialContext();
     DataSource ds =
      (DataSource) ic.lookup("java:comp/env/MyMCF");
     Connection conn = null;
     UserTransaction ut = null;

     try {
      ut = this.mySessionCtx.getUserTransaction();
         ut.begin();

         conn = ds.getConnection();

            Statement st = conn.createStatement();

            // List all of the patient names in the
            // SURG ward in the ALEXANDRIA hospital
            ResultSet rs = st.executeQuery("SELECT patname from " +
                    "pcb01.hospital, ward, patient " +
                    "where hospital.hospname = 'ALEXANDRIA' " +
                    "and ward.wardname = 'SURG'");
            while (rs.next()) {
                System.out.println(rs.getString("patname"));
            }

            // Update the name of the patient with patient
            // number 0222 in ward 04 in the hospital
            // with code R1210010000A
            int updatedRecords = st.executeUpdate("UPDATE PCB01.PATIENT " +
               "SET PATNAME='UPDATED NAME' WHERE PATNUM='0222' " +
               "AND HOSPITAL_HOSPCODE='R121001000A' AND WARD_WARDNO='04'");
            System.out.println("Updated " + updatedRecords + " Record(s)");

            rs.close();
         ut.commit();
         conn.close();
     } catch (SQLException e) {
      e.printStackTrace();
         ut.rollback();
         conn.close();
     }
   }

     /**
      * getSessionContext
      */
     public javax.ejb.SessionContext getSessionContext() {
         return mySessionCtx;
     }
```

```
    /**
     * setSessionContext
     */
    public void setSessionContext(javax.ejb.SessionContext ctx) {
        mySessionCtx = ctx;
    }

    /**
     * ejbCreate
     */
    public void ejbCreate() throws javax.ejb.CreateException {
    }

    /**
     * ejbActivate
     */
    public void ejbActivate() {
    }

    /**
     * ejbPassivate
     */
    public void ejbPassivate() {
    }

    /**
     * ejbRemove
     */
    public void ejbRemove() {
    }
}
```

Related reference:

"SQL statement usage with the IMS Universal JDBC driver" on page 596

Programming with the IMS Universal JDBC driver

IMS provides a Java Database Connectivity (JDBC) driver for SQL-based database connectivity to access IMS databases over TCP/IP with the IMS Universal JDBC driver that is included in the IMS Universal drivers. The IMS Universal JDBC driver is based on the JDBC 3.0 standard.

JDBC is an application programming interface (API) that Java applications use to access relational databases or tabular data sources. The JDBC API is the industry standard for database-independent connectivity between the Java programming language and any database that has implemented the JDBC interface. The client uses the interface to query and update data in a database. It is the responsibility of the JDBC driver itself to implement the underlying (specific) access protocol for the specific database the driver is implemented for. Drivers are client-side adapters (they are installed in the client machine, not in the server) that convert requests from Java programs to a protocol that the database management system (DBMS) can understand.

IMS support for JDBC lets you write Java applications that can issue dynamic SQL calls to access IMS data and process the result set that is returned in tabular format. The IMS Universal JDBC driver is designed to support a subset of the SQL syntax with functionality that is limited to what the IMS database management system can process natively. Its DBMS-centric design allows the IMS Universal JDBC driver to fully leverage the high performance capabilities of IMS. The IMS Universal JDBC driver also provides aggregate function support, and ORDER BY and GROUP BY support.

You can receive incoming XML documents and store them in IMS databases using the IMS Universal JDBC driver. You can also use the IMS Universal JDBC driver to compose XML documents from existing information that is stored in IMS databases.

Preparing to write a Java application with the IMS Universal drivers

Java application programs that use the IMS Universal drivers require the Java Development Kit 5.0 (JDK 5.0). Java programs that run in JMP and JBP regions require the Java Development Kit 6.0 (JDK 6.0) or later. Java application programs that use the IMS Universal drivers also require a way to generate the IMS database metadata, such as using the IMS Enterprise Suite Explorer for Development. The default segment encoding of the database metadata class produced by the IMS Explorer for Development is `cp1047`. To change the segment encoding, use the com.ibm.ims.base.DLIBaseSegment.setDefaultEncoding method.

Related concepts:

XML storage in IMS databases (Database Administration)

Related reference:

IBM Java development kits on the developerWorks website

Supported drivers for JDBC

The IMS Universal JDBC driver supports the type-2 and type-4 JDBC architectures.

The table below lists the IMS support available for the four types of JDBC driver architectures:

Table 93. JDBC driver architectures supported by IMS

JDBC driver architecture	Description	IMS support
Type-1	Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability.	IMS does not support a type-1 driver.
Type-2	Drivers that are written partly in the Java programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Because of the native code, their portability is limited. Java programs with type 2 JDBC connectivity can run on the same z/OS system or zSeries logical partition (LPAR) as the target IMS subsystem.	Use the IMS Universal JDBC driver with type-2 connectivity to access IMS from WebSphere Application Server for z/OS, IMS Java Dependent Regions (JDRs), and CICS.

Table 93. JDBC driver architectures supported by IMS (continued)

JDBC driver architecture	Description	IMS support
Type-3	Drivers that use a pure Java client and communicate with a server using a database-independent protocol. The server then communicates the client's requests to the data source.	IMS does not support a type-3 driver.
Type-4	Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.	Use the IMS Universal JDBC driver with type-4 connectivity to access the IMS subsystem via a TCP/IP network connection.

Connecting to IMS using the IMS Universal JDBC driver

You must first establish a connection to an IMS database before you can start sending queries and receiving results in your JDBC program.

Connecting to an IMS database using the JDBC DataSource interface

Using the DataSource interface is the preferred way to connect to IMS from your IMS Universal JDBC driver application.

To create and use a DataSource interface in your IMS Universal JDBC driver application:

1. Create an instance of the com.ibm.ims.jdbc.IMSDataSource class. This class is the IMS Universal JDBC driver implementation of the DataSource interface.
2. Set the following connection properties of the DataSource instance.

DatastoreName

The name of the IMS data store to access.

- When using type-4 connectivity, the **DatastoreName** property must match either the name of the data store defined to ODBM or be blank. The data store name is defined in the ODBM CSLDCxxx PROCLIB member using either the DATASTORE(NAME=*name*) or DATASTORE(NAME=*name*, ALIAS(NAME=*aliasname*)) parameter. If an alias is specified, you must specify the *aliasname* as the value of the **datastoreName** property. If the **DatastoreName** value is left blank (or not supplied), IMS Connect connects to any available instance of ODBM as it is assumed that data sharing is enabled between all datastores defined to ODBM.
- When using type-2 connectivity, set the **DatastoreName** property to the IMS subsystem alias. This is not required to be set for the Java Dependent Region run time.

MetadataURL

The location of the database metadata representing the target IMS database.

The **MetadataURL** property is the fully qualified name of the Java metadata class generated by the IMS Enterprise Suite Explorer for Development. The URL must be prefixed with `class://` (for example, `class://com.foo.BMP255DatabaseView`).

In a J2C Connection Factory environment, the **MetadataURL** property can be overridden for an individual connection without affecting the default value specified for the resource adapter.

PortNumber
The TCP/IP server port number to be used to communicate with IMS Connect. The port number is defined using the DRDAPORT parameter on the ODACCESS statement in the IMS Connect configuration PROCLIB member. The default port number is 8888. Do not set this property when using type-2 connectivity.

DatastoreServer
The name or IP address of the data store server (IMS Connect). You can provide either the host name (for example, dev123.svl.ibm.com) or the IP address (for example, 192.166.0.2). Do not set this property when using type-2 connectivity.

DriverType
The type of driver connectivity to use (value must be `IMSDataSource.DRIVER_TYPE_4` for type-4 connectivity or `IMSDataSource.DRIVER_TYPE_2` for type-2 connectivity).

user The user name for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

password
The password for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

sslConnection
Optional. Indicates if this connection uses Secure Sockets Layer (SSL) for data encryption. Set this property to "true" to enable SSL, or to "false" otherwise. Do not set this property when using type-2 connectivity.

loginTimeout
Optional. Specifies the number of seconds that the driver waits for a response from the server before timing out a connection initialization or server request. Set this property to a non-negative integer for the number of seconds. Set this property to 0 for an infinite timeout length. Do not set this property when using type-2 connectivity.

3. Optional: Set additional connection properties with the setProperties method of the IMSDataSource object.

currentSchema
Optional. Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements.

dbViewLocation
Optional. Specifies the fully qualified path to a databaseView metadata class. You can use this property to include a metadata class that is not located in your project path.

dpsbOnCommit
Optional. Set this property to **true** to deallocate the PSB when a commit occurs.

Recommendation: Do not set this property to **true** except in a managed environment with integrated connection pooling.

fetchSize
Optional. Gives the client a hint about the number of rows to get from the database when more rows are needed. The number specified for this property only affects data retrieved with the current connection. If the value specified is 0, all of the applicable rows are returned.

The default value for this property is 1 for unmanaged connections and 0 for managed connections.

llField Optional. Setting this property to **true** exposes the LL field data as a normal column in the result set. You can modify the LL field value to change the length of a variable length segment instance.

maxRows
Optional. Specifies the maximum number of rows to return in a query result set. The default value is 0, which returns all of the applicable rows in the result set.

traceFile
Optional. Specifies the name of the trace file for the connection.

traceFileAppend
Optional. If the specified trace file exists, setting this property to true specifies that the trace data for the new connection must be appended to the existing trace file instead of overwriting it.

This property is ignored if no value is specified for **traceFile**.

traceDirectory
Optional. Specifies the file system directory where the trace file is located. By default, this path is the directory where the application is executed.

This property is ignored if no value is specified for **traceFile**.

traceLevel
Optional. Specifies which traces are enabled for the connection. The valid values for this property are defined in the Java API documentation for the IMSDataSource class.

By default, all traces are disabled.

This property is ignored if no value is specified for **traceFile**.

4. Establish a connection to the data source by calling the getConnection method on the DataSource object.
5. After your application has finished with the connection, close the connection using the close method on the Connection interface.

The following code example shows how to create a type-4 connection to an IMS database from your IMS Universal JDBC driver application using the DataSource interface:

```
import java.sql.*;
import javax.sql.*;
import com.ibm.ims.jdbc.*;

Connection conn = null;

// Create an instance of DataSource
IMSDataSource ds = new com.ibm.ims.jdbc.IMSDataSource();

// Set the URL of the fully qualified name of the Java metadata class
ds.setMetadataURL("class://BMP255.BMP255DatabaseView");
```

```
// Set the data store name
ds.setDatastoreName("IMS1");

// Set the data store server
ds.setDatastoreServer("ecdev47.svl.ibm.com");

// Set the port number
ds.setPortNumber(5555);

// Set the JDBC connectivity driver typ
ds.setDriverType(IMSDataSource.DRIVER_TYPE_4);

// Enable SSL for connection
ds.setSSLConnection("true");

// Set timeout for connection
ds.setLoginTimeout("10");

// Set user ID for connection
ds.setUser("myUserID");

// Set password for connection
ds.setPassword("myPassword");

// Create JDBC connection
conn = ds.getConnection();
```

Alternatively, you can set all of the connection properties as key value pairs in a Properties object and then set them simultaneously with the IMSDataSource.setProperties method:

```
Properties props = new Properties();
props.put( "user", "MyUserID" );
props.put( "password", "MyPassword" );

IMSDataSource ds = new com.ibm.ims.jdbc.IMSDataSource();
ds.setProperties(props);
```

The typical usage of a DataSource object is for your system administrator to create and manage it separately. The program that creates and manages a DataSource object also uses the Java Naming and Directory Interface (JNDI) to assign a logical name to the DataSource object. The JDBC application that uses the DataSource object can then refer to the object by its logical name, and does not need any information about the underlying data source. In addition, your system administrator can modify the data source attributes, and you do not need to change your application program.

Recommendation: For maximum portability, use only the DataSource interface to obtain connections.

To obtain a connection using a DataSource object, given that the system administrator has already created the object and assigned a logical name to it:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Create a Context object to use in the next step. The Context interface is part of the Java Naming and Directory Interface (JNDI), not JDBC.
3. In your application program, use JNDI to get the DataSource object that is associated with the logical data source name.
4. Use the getConnection method on the DataSource instance to obtain the connection.

The following code shows an example of the code that you need in your application program to obtain a connection using a DataSource object. In this example, the logical name of the data source that you need to connect to is "jdbc/sampledb".

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/sampledb");
Connection con = ds.getConnection();
```

Related tasks:

"Configuring the IMS Universal drivers for SSL support" on page 637

Related reference:

javax.sql.DataSource methods supported (Application Programming APIs)

Connecting to an IMS database by using the JDBC DriverManager interface

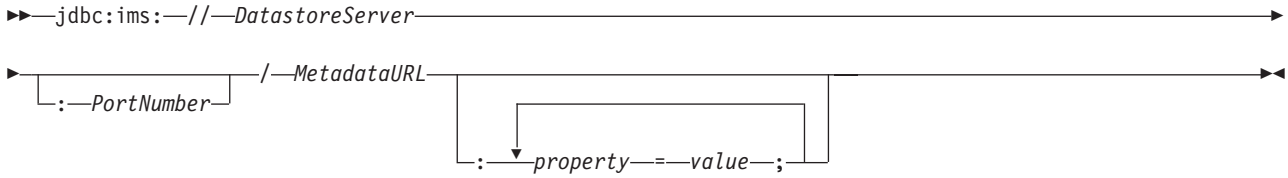
A JDBC application can establish a connection to a data source using the JDBC DriverManager interface, which is part of the java.sql package.

The Java application first loads the JDBC driver by invoking the Class.forName method. After the application loads the driver, it connects to a database server by invoking the DriverManager.getConnection method. For example:

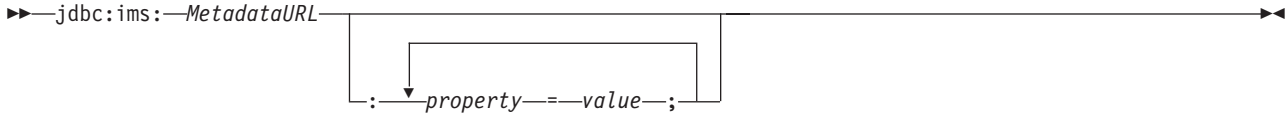
```
Connection conn = DriverManager.getConnection(url);
```

To connect to an IMS database by using the DriverManager interface in your IMS Universal JDBC driver application:

1. Load the IMS Universal JDBC driver with the DriverManager interface by invoking the Class.forName method with the argument com.ibm.ims.jdbc.IMSDriver.
2. Connect to the IMS database by invoking the DriverManager.getConnection method. The URL represents a data source, and indicates what type of JDBC connectivity you are using.
   - For type-4 connectivity, specify the URL in the following form:



   - For type-2 connectivity, specify the URL in the following form:



The parts of the URL have the following meaning:

jdbc:ims:
Indicates that the connection is to an IMS database.

PortNumber
The TCP/IP server port number to be used to communicate with IMS Connect. The port number is defined using the DRDAPORT parameter on the ODACCESS statement in the IMS Connect configuration PROCLIB member. The default port number is 8888. Do not set this property when using type-2 connectivity.

MetadataURL

The location of the database metadata representing the target IMS database.

The **MetadataURL** property is the fully qualified name of the Java metadata class generated by the IMS Enterprise Suite Explorer for Development. The URL must be prefixed with `class://` (for example, `class://com.foo.BMP255DatabaseView`).

In a J2C Connection Factory environment, the **MetadataURL** property can be overridden for an individual connection without affecting the default value specified for the resource adapter.

DatastoreServer
The name or IP address of the data store server (IMS Connect). You can provide either the host name (for example, dev123.svl.ibm.com) or the IP address (for example, 192.166.0.2). Do not set this property when using type-2 connectivity.

property
One of the following connection properties:

datastoreName

Optional. The name of the IMS data store to access.

- When using type-4 connectivity, the **DatastoreName** property must match either the name of the data store defined to ODBM or be blank. The data store name is defined in the ODBM CSLDCxxx PROCLIB member using either the DATASTORE(NAME=*name*) or DATASTORE(NAME=*name*, ALIAS(NAME=*aliasname*)) parameter. If an alias is specified, you must specify the *aliasname* as the value of the **datastoreName** property. If the **DatastoreName** value is left blank (or not supplied), IMS Connect connects to any available instance of ODBM as it is assumed that data sharing is enabled among all datastores defined to ODBM.
- When using type-2 connectivity, set the **DatastoreName** property to the IMS subsystem alias. This is not required to be set for the Java Dependent Region run time.

loginTimeout
Optional. Specifies the number of seconds that the driver waits for a response from the server before timing out a connection initialization or server request. Set this property to a non-negative integer for the number of seconds. Set this property to 0 for an infinite timeout length. Do not set this property when using type-2 connectivity.

password
The password for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

sslConnection
Optional. Indicates if this connection uses Secure Sockets Layer (SSL) for data encryption. Set this property to "true" to enable SSL, or to "false" otherwise. Do not set this property when using type-2 connectivity.

user The user name for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

currentSchema
Optional. Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements.

dbViewLocation
Optional. Specifies the fully qualified path to a databaseView metadata class. You can use this property to include a metadata class that is not located in your project path.

dpsbOnCommit
Optional. Set this property to **true** to deallocate the PSB when a commit occurs.

Recommendation: Do not set this property to **true** except in a managed environment with integrated connection pooling.

fetchSize
Optional. Gives the client a hint about the number of rows to get from the database when more rows are needed. The number specified for this property only affects data retrieved with the current connection. If the value specified is 0, all of the applicable rows are returned.

The default value for this property is 1 for unmanaged connections and 0 for managed connections.

llField Optional. Setting this property to **true** exposes the LL field data as a normal column in the result set. You can modify the LL field value to change the length of a variable length segment instance.

maxRows
Optional. Specifies the maximum number of rows to return in a query result set. The default value is 0, which returns all of the applicable rows in the result set.

traceFile
Optional. Specifies the name of the trace file for the connection.

traceFileAppend
Optional. If the specified trace file exists, setting this property to true specifies that the trace data for the new connection must be appended to the existing trace file instead of overwriting it.

This property is ignored if no value is specified for **traceFile**.

traceDirectory
Optional. Specifies the file system directory where the trace file is located. By default, this path is the directory where the application is executed.

This property is ignored if no value is specified for **traceFile**.

traceLevel
Optional. Specifies which traces are enabled for the connection. The valid values for this property are defined in the Java API documentation for the IMSDataSource class.

By default, all traces are disabled.

This property is ignored if no value is specified for **traceFile**.

value A valid value for the connection property.

To set the **sslConnection** and **loginTimeout** properties, use a java.util.Properties object. For example, the following sample code shows how to enable SSL and set the timeout value to 10 seconds:

```
Properties props = new Properties();
props.put("sslConnection", "true");
props.put("timeout", "10");
```

3. For type-4 connectivity, you must specify a user ID and password in one of the following ways: through the connection URL, through parameters, or through a java.util.Properties object. To set the user ID and password for the connection through parameters, use the form of the getConnection method that specifies *user* and *password*. For example:

```
String url =
"jdbc:ims://tst.svl.ibm.com:8888/class://BMP2.BMP2DatabaseView";
String user = "MyUserID";
String password = "MyPassword";
Connection conn = DriverManager.getConnection(url, user, password);
```

To set the user ID and password for the connection through a java.util.Properties object, use the form of the getConnection method that specifies a java.util.Properties object. For example:

```
Properties props = new Properties();
props.put( "user", "MyUserID" );
props.put( "password", "MyPassword" );
String url =
"jdbc:ims://tst.svl.ibm.com:8888/class://BMP2.BMP2DatabaseView";
Connection conn = DriverManager.getConnection(url, props);
```

4. After your application has finished with the connection, close the connection using the close method on the Connection interface.

The following code example shows how to create a type-4 connection to an IMS database from your IMS Universal JDBC driver application using the DriverManager interface:

```
Connection conn = null;

// Create Properties object
Properties props = new Properties();

// Enable SSL for connection
props.put("sslConnection", "true");

// Set datastoreName for connection
props.put( "datastoreName", "IMS1" );

// Set timeout for connection
props.put("loginTimeout", "10");

// Set user ID for connection
props.put( "user", "myUserID" );

// Set password for connection
```

```
props.put( "password", "myPassword" );

// Set URL for the data source
Class.forName("com.ibm.ims.jdbc.IMSDriver");

// Create connection
conn = DriverManager.getConnection
("jdbc:ims://tst.svl.ibm.com:8888/class://BMP2.BMP2DatabaseView",
props);
```

Alternatively, you can specify the connection properties in the URL. For example:

```
String url=""jdbc:ims://tst.svl.ibm.com:8888/class://"
   + "BMP2.BMP2DatabaseView:datastoreName=IMS1;"
   + "loginTimeout=10;sslConnection=true;user=myUserID;password=myPassword;";
Connection conn = DriverManager.getConnection(url);
```

Related tasks:

"Configuring the IMS Universal drivers for SSL support" on page 637

Sample application for the IMS Universal JDBC driver

The following sample Java application demonstrates the basic programming flow for a JDBC application using the IMS Universal JDBC driver.

The following example connects to an IMS database, retrieves a list of patient names using a SQL SELECT query, and modifies the patient information using a SQL UPDATE query.

```
package client;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.ibm.ims.jdbc.IMSDataSource;

public class JDBCSample {


    public static void main(String[] args)
       throws SQLException {
        IMSDataSource ds = new IMSDataSource();
        ds.setMetadataURL("class://MyDatabaseView");
        ds.setDatastoreName("IMS1");
        ds.setDatastoreServer("ec0123.my.host.com");
        ds.setPortNumber(5555);
        ds.setDriverType(IMSDataSource.DRIVER_TYPE_4);
        ds.setUser("myUserId");
        ds.setPassword("myPassword");

        Connection conn = null;

        try {
            conn = ds.getConnection();

            Statement st = conn.createStatement();

            // List all of the patient names in the
            // SURG ward in the ALEXANDRIA hospital
            ResultSet rs = st.executeQuery("SELECT patname from " +
                    "pcb01.hospital, ward, patient " +
                    "where hospital.hospname = 'ALEXANDRIA' " +
                    "and ward.wardname = 'SURG'");
            while (rs.next()) {
                System.out.println(rs.getString("patname"));
```

```
                }

                // Update the name of the patient with patient
                // number 0222 in ward 04 in the hospital
                // with code R1210010000A
                int updatedRecords = st.executeUpdate("UPDATE PCB01.PATIENT " +
                   "SET PATNAME='UPDATED NAME' WHERE PATNUM='0222' " +
                   "AND HOSPITAL_HOSPCODE='R121001000A' AND WARD_WARDNO='04'");
                System.out.println("Updated " + updatedRecords + " Record(s)");

                conn.commit();
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
                if (!conn.isClosed()) {
                    conn.rollback();
                    conn.close();
                }
            }
        }
    }
```

Writing SQL queries to access an IMS database with the IMS Universal JDBC driver

Use the IMS Universal JDBC driver to connect to an IMS database for writing SQL queries.

The IMS Explorer for Development generates a Java database metadata class. This class that contains the PSB and DBD metadata classes that the IMS Universal drivers use at runtime.

The metadata includes information about the IMS database, including segments, segment names, the segment hierarchy, fields, field types, field names, fields offsets, and field lengths.

The metadata is used by the IMS Universal JDBC driver to allocate program specification blocks (PSBs), issue DL/I calls, perform data transformation, and translate SQL queries to DL/I calls.

The following table summarizes the mapping between IMS database elements and relational database elements.

Table 94. Mapping between IMS database elements and relational database elements.

Hierarchical database elements in IMS	Equivalent relational database elements
Segment name	Table name
Segment instance	Table row
Segment field name	Column name
Segment unique key	Table primary key
Foreign key field	Table foreign key

Related concepts:

IMS Explorer for Development overview

SQL keywords supported by the IMS JDBC drivers

The SQL support provided by the IMS classic JDBC driver and the IMS Universal JDBC driver is based on the SQL-92 standard for relational database management systems.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

If you use a SQL keyword as a name for a PCB, segment, or field, your JDBC application program will throw an error when it attempts an SQL query. These keywords are not case-sensitive.

The following SQL keywords are supported by the IMS JDBC drivers:

ALL
AND
AS
ASC
AVG
COUNT
DELETE
DESC
DISTINCT
FROM
GROUP BY
INSERT
INTO
MAX
MIN
OR
ORDER BY
SELECT
SET
SUM
UPDATE
VALUES
WHERE

Related reference:

"SELECT statement usage" on page 690

"FROM clause usage" on page 692

"WHERE clause usage" on page 693

"Portable SQL keywords restricted by the IMS Universal JDBC drivers" on page 595

SQL aggregate functions supported by the IMS JDBC drivers

The IMS classic JDBC driver and the IMS Universal JDBC driver support SQL aggregate functions and related keywords.

- AS
- AVG
- COUNT
- GROUP BY
- MAX

- MIN
- ORDER BY
  - ASC
  - DESC
- SUM

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

Restriction: The supported SQL aggregate functions accept only a single field name in a segment as the argument (the DISTINCT keyword is not allowed). The exception is the COUNT function, which allows the DISTINCT keyword. This restriction applies to the IMS classic JDBC driver, but not the IMS Universal JDBC driver.

The ResultSet type for aggregate functions and ORDER BY and GROUP BY clauses is always TYPE_SCROLL_INSENSITIVE.

The following table shows the data types of the fields that are accepted by the aggregate functions, along with the resulting data type in the ResultSet.

Table 95. Supported SQL aggregate functions and their supported data types

Function	Argument type	Result type
SUM and AVG	Byte	Long
	Short	Long
	Integer	Long
	Long	Long
	BigDecimal	Double-precision floating point
	Single-precision floating point	Double-precision floating point
	Double-precision floating point	Double-precision floating point
MIN and MAX	Any type except BIT, BLOB, or BINARY	Same as argument type
COUNT	Any type	Long

Column names generated by aggregate functions

The ResultSet column name from an aggregate function is a combination of the aggregate function name and the field name separated by an underscore character (_). For example, the statement `SELECT MAX(age)` results in a column name MAX_age. Use this column name in all subsequent references—for example, `resultSet.getInt("MAX_age")`.

If the aggregate function argument field is table-qualified, the ResultSet column name is the combination of the aggregate function name, the table name, and the column name, separated by underscore characters (_). For example, `SELECT MAX(Employee.age)` results in a column name MAX_Employee_age.

Using the AS clause

You can use the AS keyword to rename the aggregate function column in the result set or any other field in the SELECT statement. You cannot use the AS keyword to rename a table in the FROM clause. When you use the AS keyword to rename the column, you must use this new name to refer to the column. For example, if you specify `SELECT MAX(age) AS oldest`, a subsequent reference to the aggregate function column is `resultSet.getInt("oldest")`.

If you are using the IMS Universal JDBC driver and you specified a SELECT query with column names renamed by an AS clause, you can only refer to the field in the resulting ResultSet by the AS rename. However, in the rest of your SELECT query, in the WHERE, ORDER BY, and GROUP BY clauses, you can use either the original column name or the AS rename.

Using the ORDER BY and GROUP BY clauses

Important: The field names that are specified in a GROUP BY or ORDER BY clause must match exactly the field name that is specified in the SELECT statement.

When using the IMS Universal JDBC driver, the following queries with the ORDER BY and GROUP BY clauses are valid:

```
SELECT HOSPNAME, COUNT(PATNAME) AS PatCount FROM PCB01.HOSPITAL, PATIENT
GROUP BY HOSPNAME ORDER BY HOSPNAME
```

```
SELECT HOSPNAME, COUNT(DISTINCT PATNAME) AS PatCount FROM PCB01.HOSPITAL,
PATIENT GROUP BY HOSPNAME ORDER BY HOSPNAME
```

Using the COUNT function with DISTINCT

When using the IMS Universal JDBC driver, the COUNT aggregate function can be qualified with the DISTINCT keyword. For example, the following query returns all hospital names listed in ascending order along with the number of distinct patient names from that hospital. The COUNT aggregate function generates a column name COUNT_DISTINCT_PATNAME .

```
SELECT HOSPNAME, COUNT(DISTINCT PATNAME)FROM PCB01.HOSPITAL, PATIENT GROUP
BY HOSPNAME ORDER BY HOSPNAME
```

Portable SQL keywords restricted by the IMS Universal JDBC drivers

If you use any of the following SQL keywords as a name for a PCB, segment, or field, your JDBC application will receive an error when it attempts an SQL query. Instead, use the aliasing feature of the IMS Enterprise Suite DLIModel utility plug-in to provide an alias for the restricted name. These keywords are not case-sensitive.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

The keywords shown in the following table are reserved SQL keywords.

ABORT to COT	CURRENT to IS	JOIN to REAL	REFERENCES to WORK
ABORT	CURRENT	JOIN	REFERENCES
ABS	CREATE	LAST	RESET
ACOS	CROSS	LEADING	REVOKE
ANALYZE	CURRENT	LEFT	RIGHT
AND	CURSOR	LIKE	ROLLBACK
ALL	DECIMAL	LISTEN	ROW
ALLOCATE	DECLARE	LN	ROWS
ALTER	DEFAULT	LOAD	SELECT
AND	DEGREES	LOCAL	SET
ANY	DELETE	LOCK	SETOF
ARE	DESC	LOG	SHOW
AS	DISTINCT	LOG10	SIGN
ASC	DO	MAX	SIN
ASSERTION	DOUBLE	MIN	SINH
ASIN	DROP	MOD	SMALLINT
AT	END	MOVE	SQRT
ATAN	EXECUTE	NAMES	SUBSTRING
ATAN2	EXISTS	NATIONAL	SUM
AVG	EXP	NATURAL	TABLE
BEGIN	EXPLAIN	NCHAR	TAN
BETWEEN	EXTRACT	NEW	TANH
BINARY	EXTEND	NO	TO
BIT	FALSE	NONE	TRAILING
BOOLEAN	FETCH	NOT	TRANSACTION
BOTH	FIRST	NOTIFY	TRIM
BY	FLOAT	NULL	TRUE
CASCADE	FLOOR	NUMERIC	UNION
CAST	FOR	ON	UNIQUE
CEIL	FOREIGN	ONLY	UNLISTEN
CEILING	FROM	OR	UNTIL
CHAR	FULL	ORDER	UPDATE
CHARACTER	GRANT	OUTER	USER
CHECK	GROUP	PARTIAL	USING
CLOSE	HAVING	POSITION	VACCUM
CLUSTER	IN	POWER	VALUES
COLLATE	INNER	PRECISION	VARCHAR
COLUMN	INSERT	PRIMARY	VARYING
COMMIT	INT	PRIVILEGES	VERBOSE
CONSTRAINT	INTEGER	PROCEDURE	VIEW
COPY	INTERVAL	PUBLIC	WHERE
COS	INTO	RADIANS	WITH
COSH	IS	REAL	WORK
COT			

Related reference:

"SQL keywords supported by the IMS JDBC drivers" on page 593

SQL statement usage with the IMS Universal JDBC driver

The following usage rules apply to SQL statements passed to IMS with the IMS Universal JDBC driver.

Foreign key fields:

In relational databases, hierarchies can be logically built by creating foreign key relationships between tables. In IMS, the hierarchies are explicit and are part of the database definition itself. The IMS Universal JDBC driver introduces the concept of *foreign keys* to capture these explicit hierarchies in a relational sense, which makes the SQL syntax for IMS equivalent to standard SQL.

When accessing IMS databases with the IMS Universal JDBC driver, every table that is not the root table in a hierarchic path will virtually contain the unique keys of all of its parent segments up to the root of the database. These keys are called *foreign key fields*.

The purpose of the foreign key fields is to maintain referential integrity, similar to foreign keys in relational databases. This allows SQL SELECT, INSERT, UPDATE, and DELETE queries to be written against specific tables and columns located in a hierarchic path.

Remember: Foreign keys are maintained internally by the IMS Universal JDBC driver; the keys are not physically stored in the IMS database.

Hospital database example without a secondary index

For example, in the Hospital database, the HOSPITAL, WARD, and PATIENT tables are on the same hierarchic path. The JDBC application would view the tables as containing the following columns.

HOSPITAL table

Columns are:

- HOSPNAME
- HOSPCODE (primary key)

WARD table

Columns are:

- WARDNO (primary key)
- WARDNAME
- PATCOUNT
- NURCOUNT
- DOCCOUNT
- HOSPITAL_HOSPCODE (foreign key field referencing the HOSPCODE column in the HOSPITAL table)

PATIENT table

Columns are:

- PATNUM (primary key)
- PATNAME
- WARD_WARDNO (foreign key field referencing the WARDNO column in the WARD table)
- HOSPITAL_HOSPCODE (foreign key field referencing the HOSPCODE column in the HOSPITAL table)

The following queries show how SQL SELECT statements can use foreign keys, based on the previous database example. The following statement retrieves all columns from a PATIENT table derived from a child segment under HOSPITAL and WARD on a hierarchic path:

```
SELECT * FROM PCB01.PATIENT
WHERE HOSPITAL_HOSPCODE = 'H5140070000H'
   AND WARD_WARDNO = '0023'
```

The following example shows an INSERT statement using foreign keys:

```
INSERT INTO PCB01.PATIENT (PATNUM, PATNAME,
WARD_WARDNO, HOSPITAL_HOSPCODE)
VALUES ('00345', 'John Doe', '0023', 'H514007000H')
```

The following statements retrieve the hospital code and all ward names from a WARD table. These statements are all equivalent:

```
SELECT HOSPITAL.HOSPCODE, WARD.WARDNAME
FROM PCB01.HOSPITAL, PCB01.WARD
```

```
SELECT HOSPITAL_HOSPCODE, WARD.WARDNAME
FROM PCB01.WARD
```

```
SELECT WARD.HOSPITAL_HOSPCODE, WARD.WARDNAME
FROM PCB01.WARD
```

```
SELECT HOSPITAL_HOSPCODE, WARDNAME
FROM PCB01.WARD
```

The following statement will **fail** because the column HOSPITAL_HOSPCODE is not in the table HOSPITAL.:

```
SELECT HOSPITAL_HOSPCODE FROM PCB01.HOSPITAL
```

SELECT statement usage:

The SELECT statement is used to retrieve data from one or more tables. The result is returned in a tabular result set.

When using the SELECT statement with the IMS Universal JDBC driver:

- If you are selecting from multiple tables and the same column name exists in one or more of these tables, you must table-qualify the column or an ambiguity error will occur.
- The FROM clause must list all the tables you are selecting data from. The tables listed in the FROM clause must be in the same hierarchic path in the IMS database.
- In Java applications using the IMS JDBC drivers, connections are made to PSBs. Because there are multiple database PCBs in a PSB, queries must specify which PCB in a PSB to use. To specify which PCB to use, always qualify segments that are referenced in the FROM clause of an SQL statement by prefixing the segment name with the PCB name. You can omit the PCB name only if the PSB contains only one PCB.

Examples of valid IMS Universal JDBC driver SELECT queries

Selecting specified columns

The following statement retrieves the ward names and patient names from the WARD and PATIENT tables, respectively:

```
SELECT WARD.WARDNAME,PATIENT.PATNAME
FROM PCB01.WARD, PATIENT
```

Selecting all columns with * symbol

The following statement retrieves all columns for the PATIENT table:

```
SELECT *
FROM PCB01.PATIENT
```

The following statement retrieves the hospital name from the HOSPITAL table and all columns from the WARD table:

```
SELECT HOSPITAL.HOSPNAME, WARD.*
FROM PCB01.HOSPITAL, PCB01.WARD
```

Selecting with DISTINCT

The following statement retrieves all distinct patient names from the PATIENT table:

```
SELECT DISTINCT PATNAME
FROM PCB01.PATIENT
```

Selecting with ORDER BY

The ORDER BY clause is used to sort the rows. By default, results are sorted by ascending numerical or alphabetical order. The following statement retrieves all distinct hospital names, sorted in alphabetical order:

```
SELECT DISTINCT HOSPNAME FROM PCB01.HOSPITAL
   ORDER BY HOSPNAME
```

The following statement retrieves all ward names sorted in alphabetical order, and the number of patients in each ward sorted in ascending numerical order. If two WARDNAME values in the ORDER BY compare are equal, the tiebreaker will be their corresponding PATCOUNT values (in this case, the row with the numerically smaller corresponding PATCOUNT value is displayed first).

```
SELECT WARDNAME, PATCOUNT FROM PCB01.WARD
   ORDER BY WARDNAME, PATCOUNT
```

Use the DESC qualifier to sort the query result in descending numerical or reverse alphabetical order. The following statement retrieves all patient names in reverse alphabetical order:

```
SELECT PATNAME FROM PCB01.PATIENT
   ORDER BY PATNAME DESC
```

Use the ASC qualifier to explicitly sort the query result in ascending numerical or reverse alphabetical order. The following statement retrieves all ward names sorted in ascending alphabetical order, and the number of patients in each ward sorted in descending numerical order:

```
SELECT WARDNAME, PATCOUNT FROM PCB01.WARD
   ORDER BY WARDNAME ASC, PATCOUNT DESC
```

Selecting with GROUP BY

The GROUP BY clause is used to return results for aggregate functions, grouped by distinct column values. The following statement returns the aggregated sum of all doctors in every ward in a hospital, grouped by distinct ward names:

```
SELECT WARDNAME, SUM(DOCCOUNT)
FROM PCB01.WARD
WHERE HOSPITAL_HOSPCODE = 'H5140070000H
   GROUP BY WARDNAME
```

The following statement returns the hospital name, ward name, and the count of all patients in each ward in each hospital, grouped by distinct hospital names and sub-grouped by ward names:

```
SELECT HOSPNAME, WARDNAME, COUNT(PATNAME)
FROM PCB01.HOSPITAL, WARD, PATIENT
   GROUP BY HOSPNAME, WARDNAME
```

Using the AS clause

Use the AS clause to rename the aggregate function column in the result set or any other field in the SELECT statement. The following statement returns the aggregate count of distinct patients in the PATIENT table with the alias of "PATIENTCOUNT":

```
SELECT COUNT(DISTINCT PATNAME)
   AS PATIENTCOUNT
FROM PCB01.PATIENT
```

The following statement returns the aggregate count of distinct wards in all hospitals with the alias of "WARDCOUNT", sorted by the hospital names in alphabetical order, and grouped by distinct hospital names (under a renamed column alias "HOSPITALNAME"):

```
SELECT HOSPNAME AS HOSPITALNAME, COUNT(DISTINCT WARDNAME)
   AS WARDCOUNT
FROM PCB01.HOSPITAL, WARD
   GROUP BY HOSPNAME
   ORDER BY HOSPNAME
```

Examples of invalid IMS Universal JDBC driver SELECT queries

Examples of valid IMS classic JDBC driver SELECT queries that are invalid for the IMS Universal JDBC driver

The following statement is invalid because the FROM clause is missing the WARD table:

```
SELECT WARD.WARDNAME,PATIENT.PATNAME
FROM PCB01.PATIENT
```

The following statement is invalid because the FROM clause is missing the PATIENT table:

```
SELECT PATIENT.*, ILLNESS.*
FROM PCB01.ILLNESS
```

INSERT statement usage:

The INSERT statement is used to insert new rows into a table.

Foreign key fields enable the IMS Universal JDBC driver to properly position the new record (or segment instance) to be inserted in the hierarchic path using standard SQL processing, similar to foreign keys in a relational database. When inserting a record in a table at a non-root level, you must specify values for all the foreign key fields of the table.

Examples of valid IMS Universal JDBC driver INSERT statements

Inserting data at the root

The following statement inserts a new HOSPITAL record:

```
INSERT INTO PCB01.HOSPITAL (HOSPCODE, HOSPNAME)
VALUES ('R1210050000A', 'O''MALLEY CLINIC')
```

Inserting data into a specified table in a hierarchic path

When inserting a record in a table at a non-root level, you must specify values for all the foreign key fields of the table. The following statement inserts a new ILLNESS record under a specific HOSPITAL, WARD, and PATIENT table. In this example, the ILLNESS table has three foreign keys

HOSPITAL_HOSPCODE, WARD_WARDNO, and PATIENT_PATNUM. The new record will be inserted if and only if there is a HOSPCODE in the HOSPITAL table with the value of 'H5140070000H', a WARD table with a WARDNO value of '01', and a PATIENT table with PATNUM value of 'R1210050000A'.

```
INSERT INTO PCB01.ILLNESS (HOSPITAL_HOSPCODE, WARD_WARDNO,
   ILLNAME, PATIENT_PATNUM)
VALUES ('H5140070000H', '01', 'COLD', 'R1210050000A')
```

The following statement inserts a new WARD record under a specific HOSPITAL table. In this example, the WARD table has the foreign key HOSPITAL_HOSPCODE. The new record will be inserted if and only if there is a HOSPCODE in the HOSPITAL table with the value of 'H5140070000H'.

```
INSERT INTO PCB01.WARD (WARDNO, HOSPITAL_HOSPCODE, WARDNAME)
VALUES ('0001', 'H5140070000H', 'EMGY')
```

Inserting data in a searchable field with subfields

If a searchable field consists of subfields, you can insert data by setting all the subfield values such that the searchable field is completely populated.

Examples of invalid IMS Universal JDBC driver INSERT statements

Example of a valid IMS classic JDBC driver INSERT query that is invalid for the IMS Universal JDBC driver

This statement is invalid for the IMS Universal JDBC driver because it does not use the correct syntax to specify a legal value for the foreign key field (HOSPITAL_HOSPCODE). Note that it is not standard SQL to use a WHERE clause on an INSERT statement.

```
INSERT INTO PCB01.WARD (WARDNO, WARDNAME)
VALUES ('01', 'EMGY')
WHERE HOSPITAL.HOSPCODE = 'H5140070000H'
```

Inserting a record at a non-root level without specifying foreign key fields

In this statement, the WARD_WARDNO foreign key field is missing. The query will fail because it violates the referential integrity constraint that all foreign keys must be provided with legal values.

```
INSERT INTO PCB01.PATIENT (HOSPITAL_HOSPCODE, PATNAME, PATNUM)
VALUES ('HW3201', 'JOHN O''CONNER', 'Z800')
```

UPDATE statement usage:

The UPDATE statement is used to modify the data in a table.

Examples of valid IMS Universal JDBC driver UPDATE statements

Updating one column in a record

The following statement updates the root:

```
UPDATE HOSPITAL SET HOSPNAME = 'MISSION CREEK'
WHERE HOSPITAL.HOSPCODE = 'H001007'
```

Updating multiple columns in a specified record in a hierarchic path

Foreign keys allow the IMS Universal JDBC driver to maintain referential integrity by identifying the exact record (or segment instance) to update. The following statement updates a WARD record under a specific HOSPITAL. In this example, the WARD table has the foreign key HOSPITAL_HOSPCODE. The record will be updated if and only if there is a HOSPCODE in the HOSPITAL table with the value of 'H5140070000H'.

```
UPDATE WARD SET WARDNAME = 'EMGY',
   DOCCOUNT = '2', NURCOUNT = '4'
WHERE HOSPITAL_HOSPCODE = 'H5140070000H'
   AND WARDNO = '01'
```

Examples of invalid IMS Universal JDBC driver UPDATE statements

Example of a valid IMS classic JDBC driver UPDATE query that is invalid for the IMS Universal JDBC driver

This statement is invalid for the IMS Universal JDBC driver because it does not use the correct syntax to specify a legal value for the foreign key field (HOSPITAL_HOSPCODE).

```
UPDATE WARD SET WARDNAME = 'EMGY',
   DOCCOUNT = '2', NURCOUNT = '4'
WHERE HOSPITAL.HOSPCODE = 'H5140070000H'
   AND WARDNO = '01'
```

Updating a foreign key field

Making an UPDATE on a foreign key field is invalid for the IMS Universal JDBC driver. For example, the following UPDATE query will fail:

```
UPDATE WARD SET WARDNAME = 'EMGY',
   HOSPITAL_HOSPCODE = 'H5140070000H'
WHERE WARDNO = '01'
```

DELETE statement usage:

The DELETE statement is used to delete rows in a table. DELETE operations are cascaded to all child segments.

Examples of valid IMS Universal JDBC driver DELETE statements

Deleting an entire database

The following statement deletes the HOSPITAL database:

```
DELETE FROM pcb01.HOSPITAL
```

Deleting a root

The following statement deletes the root segment instance and all its children in the hierarchic path.

```
DELETE FROM pcb01.HOSPITAL
WHERE HOSPCODE = 'H5140070000H'
```

Deleting a single record

Foreign keys allow the IMS Universal JDBC driver to maintain referential integrity by identifying the exact record (or segment instance) to delete. The following statement deletes a single record from the WARD table. In this example, the WARD table has the foreign key HOSPITAL_HOSPCODE. The WARD record will be deleted if and only if there is a HOSPCODE in the HOSPITAL table with the value of 'H5140070000H' and a WARD table with a WARDNO value of '0001'.

```
DELETE FROM pcb01.WARD
WHERE HOSPITAL_HOSPCODE = 'H5140070000H'
   AND WARDNO = '0001'
```

Deleting multiple records

The following statement deletes multiple records from the PATIENT table. In this example, the PATIENT table has two foreign keys: HOSPITAL_HOSPCODE and WARD_WARDNO. The PATIENT record will be deleted if and only if there is a HOSPCODE in the HOSPITAL table with the value of 'H5140070000H', a WARD table with WARDNO value of '0001', and a PATIENT table with a PATNUM value greater than '0007'.

```
DELETE FROM pcb01.PATIENT
WHERE PATNUM > '0007'
   AND HOSPITAL_HOSPCODE = 'H5140070000H'
   AND WARD_WARDNO = '0001'
```

The following statement deletes all WARD segment instances in the entire database:

```
DELETE FROM pcb01.WARD
```

Examples of invalid IMS Universal JDBC driver DELETE statements

Example of a valid IMS classic JDBC driver DELETE query which is invalid for the IMS Universal JDBC driver

This statement is invalid for the IMS Universal JDBC driver because it does not use the correct syntax to specify a legal value for the foreign key field (HOSPITAL_HOSPCODE).

```
DELETE FROM pcb01.WARD
WHERE HOSPITAL.HOSPCODE = 'H5140070000H'
   AND WARDNO = '0001'
```

WHERE clause usage:

In a SQL SELECT, UPDATE, and DELETE statement, the WHERE clause can be used to select data conditionally.

When using the WHERE clause with the IMS Universal JDBC driver, use columns that are in any table listed in the FROM clause.

Recommendation: Qualify columns with table names. If you do not table-qualify a column, there can be ambiguity if that column exists in more than one table that was joined in the FROM clause.

The IMS JDBC drivers convert the WHERE clause in an SQL query to a segment search argument (SSA) list when querying a database. SSA rules restrict the type of conditions you can specify in the WHERE clause. The following restrictions apply:

- In general, compare columns to values, not other columns. With the introduction of foreign keys, it is legal to compare one column to another column if one column is the foreign key and the other column is the primary key it is referencing. For example:

  ```
  WHERE HOSPITAL_HOSPCODE = HOSPITAL.HOSPCODE
  ```

  You can use the following operators between column names and values in the individual qualification statements:

  = Equals

  != Not equal

  > Greater than

  >= Greater than or equals

  < Less than

  <= Less than or equals

  For example, the following WHERE clause will fail because it is trying to compare two columns:

  ```
  WHERE PAYMENTS.PATNUM=PAYMENTS.AMOUNT
  ```

The following example is valid because the WHERE clause is comparing a column to a value:

```
WHERE PAYMENTS.PATNUM='A415'
```

- Do not use parentheses. Qualification statements are evaluated from left to right. The order of evaluation for operators is the IMS evaluation order for segment search arguments.
- List all qualification statements for a table adjacently. For example, in the following valid WHERE clause, the qualified columns from the same PATIENT table are listed adjacently:

  ```
  WHERE PATIENT.PATNAME='BOB' OR PATIENT.PATNUM='A342' AND WARD.WARDNO='52'
  ```

  The following invalid WHERE clause will fail because the columns from the HOSPITAL table are separated by the columns from the WARD table:

  ```
  WHERE HOSPITAL.HOSPNAME='Santa Teresa' AND WARD.WARDNO='52'
  OR WARD.WARDNAME='CARD' AND HOSPITAL.HOSPCODE='90'
  ```

- The OR operator can be used only between qualification statements that contain columns from the same table. You cannot use the OR operator across tables. To combine qualification statements for different tables, use an AND operator. For example, the following invalid WHERE clause will fail:

  ```
  WHERE WARD.WARDNO='03' OR PATIENT.PATNUM='A415'
  ```

  However, the following WHERE clause is valid because the OR operator is between two qualification statements for the same table:

  ```
  WHERE PATIENT.PATNUM='A409' OR PATIENT.PATNAME='Sandy'
  ```

- The columns in the WHERE clause must be DBD-defined fields. These columns that are in the DBD are marked in the DLIModel IMS Java report as being either primary key fields or search fields. The only exception to this is when the columns in the WHERE clause are subfields that make up a DBD-defined field.
- When using prepared statements, you can use the question mark (?) character, which is later filled in with a value. For example, the following WHERE clause is valid:

  ```
  WHERE PAYMENTS.AMOUNT>?
  ```

WHERE clause subfield support:

When passing SQL statements using the IMS JDBC drivers, you can use the WHERE clause to list subfields of any field, as long as the field is searchable and is fully defined by the subfields.

For example, a DBD-defined field is named ADDRESS and is 30 bytes long. In a COBOL copybook, this field is broken down into CITY, STATE, and ZIPCODE subfields, as illustrated by the code below.

```
01 ADDRESS
   02 CITY PIC X(10)
   02 STATE PIC X(10)
   03 ZIP PIC X(10)
```

Without the subfield support, the ADDRESS value in the WHERE clause would have to be padded manually, and entered like this:

```
WHERE ADDRESS = 'san jose  ca        95141     '
```

With the subfield support, you can enter the WHERE clause like this:

```
WHERE CITY = 'san jose'
   AND STATE = 'ca'
   AND ZIPCODE = '95141'
```

The IMS JDBC drivers will convert the individual subfields and bundle them into the ADDRESS field before sending the SQL query to IMS.

The following usage rules and restrictions apply to WHERE clause subfield support:

- Parameter markers are supported for subfields. For example, for a prepared statement, the following WHERE clause entry is valid:

```
WHERE CITY = ? AND STATE = ? AND ZIPCODE = ?
```

- The only relational operator supported for subfields is "=" (equals operator).
- The only Boolean operator is "AND" for connecting subfields. The following WHERE clause entry is valid because the subfields are connected using only "AND" operators:

```
WHERE HOSPCODE=? OR CITY = ? AND STATE = ? AND ZIPCODE = ?
```

- All the subfields for a particular searchable field must be specified in the WHERE clause. You cannot omit any subfields of a field. For example, the following WHERE clause entry is invalid because the STATE subfield was not provided:

```
WHERE CITY = ? AND ZIPCODE = ?
```

- When specifying the subfields in a WHERE clause, all the subfields for a searchable field must be listed adjacent to each other. For example, the following WHERE clause entry is invalid because the listing of the subfields is not contiguous:

```
WHERE CITY = ? AND STATE = ? OR HOSPCODE=? AND ZIPCODE = ?
```

- You can enter subfields for multiple searchable fields in the WHERE clause. For example, if the PATNAME field was broken into LASTNAME and FIRSTNAME subfields, you can specify the subfields for ADDRESS and PATNAME as follows:

```
WHERE CITY = ? AND STATE = ? AND ZIPCODE = ?
   OR LASTNAME = ? AND FIRSTNAME = ?
```

- When specifying the subfields in a WHERE clause across multiple tables, all the subfields for the searchable fields in each table must be listed together, before listing the subfields for the next table. For example, if the ADDRESS field was in the HOSPITAL table and the PATNAME field was in the PATIENT table, the following WHERE clause entry is invalid because not all the ADDRESS subfields have been listed for HOSPITAL:

```
WHERE HOSPITAL.CITY = ? AND HOSPITAL.ZIPCODE = ?
   AND PATIENT.LASTNAME = ? AND PATIENT.FIRSTNAME = ?
```

IMS Universal JDBC driver support for XML

You can write applications to store XML data in IMS databases or retrieve XML data from IMS databases by using the IMS Universal JDBC driver. Both the type-4 and type-2 drivers offer this support.

You can use the IMS Universal JDBC driver support for XML to complete the following operations:

- Retrieve XML data from an IMS database as a character large object (CLOB) through a SQL SELECT statement.
- Store XML data into an IMS database, through a SQL INSERT statement, by using either the PreparedStatement.setClob method or the PreparedStatement.setCharacterStream method.

The syntax for storing and retrieving XML data by using the IMS Universal JDBC driver is independent of how the XML data is physically stored in the IMS database. The interface is not sensitive to whether the data is stored in

decomposed storage mode, intact storage mode, or both or whether the data is stored in an existing or new IMS database.

To use IMS Universal JDBC driver support for XML, you need to generate a runtime Java metadata class that corresponds to a program specification block (PSB). The Java metadata class must define the column fields in the IMS database for storing and retrieving XML data and identify the XML schema that describes the structure of the data.

Defining XML datatype column fields in the Java metadata class

To use IMS Universal JDBC driver support for XML, you need to define the XML datatype column fields for storing and retrieving XML data.

To define an XML data type column field in the Java metadata class:

1. Generate the Java metadata class with the IMS Enterprise Suite Explorer for Development.
2. Generate the XML schema for the database with the IMS Enterprise Suite DLIModel utility plug-in or manually create the XML schema based on the DBD and data type mappings.
3. Specify the XML datatype column field by modifying the generated Java metadata class. If you are storing or retrieving XML data in decomposed storage mode, define the XML datatype column field with the following DLITypeInfo constructor syntax. One or more XML datatype column fields can be defined in a segment.

   ```
   public DLITypeInfo(String fieldName,
               String XMLSchemaName,
                DLITypeInfo.XML);
   ```

4. During database connection setup, pass the name of the Java metadata class to the IMS Universal JDBC driver.

The following example shows how to define XML column datatype fields in a Java metadata class for decomposed mode. In this example, an XML datatype column field named “HOSPXML” is defined that is associated with the “BMP255-PCB01.xsd” XML schema. Another XML datatype column field named “HXML” is defined that is associated with the “B.xsd” XML schema.

```
// The following describes Segment: HOSPITAL ("HOSPITAL") in PCB: PCB01 ("PCB01")
    static DLITypeInfo[] PCB01HOSPITALArray= {
        new DLITypeInfo("HOSPLL", DLITypeInfo.CHAR, 1, 2, "HOSPLL"),
        new DLITypeInfo("HOSPCODE", DLITypeInfo.CHAR, 3, 12,
           "HOSPCODE", DLITypeInfo.UNIQUE_KEY),
        new DLITypeInfo("HOSPNAME", DLITypeInfo.CHAR, 15, 17, "HOSPNAME"),
        new DLITypeInfo("HOSPXML", "BMP255-PCB01.xsd", DLITypeInfo.XML),
        new DLITypeInfo("HXML", "B.xsd", DLITypeInfo.XML)
    };
```

Storing XML data by using the IMS Universal JDBC driver

You can use the IMS Universal JDBC driver to store XML data into an IMS database through an SQL INSERT statement.

To store XML data in your IMS Universal JDBC driver application:

1. Specify the file path that contains the XML schema file (.xsd) that describes the input XML data structure by setting the *http://www.ibm.com/ims/schema-resolver/file/path* environment variable. The following example shows how to programmatically set the environment variable. In this example, the file path `uxml/samples` indicates a relative path to the XML schema file. You can also specify an absolute file path.

```
System.setProperty("http://www.ibm.com/ims/schema-resolver/file/path",
   "uxml/samples");
```

2. Specify your XML data source. If you are reading in XML data from an external source, such as a file, you must create a java.io.Reader object to wrap the input XML data. The following example shows how to create an InputStreamReader object to wrap an external file named `hospwashington.xml`. The InputStreamReader object converts the bytes that are read from the input file from ASCII encoding to Unicode.

```
String doc = "hospwashington.xml";
InputStream fileStream = getClass().getResourceAsStream(doc);
if (fileStream == null) {
    throw new FileNotFoundException("Insert Document: '" + doc + "' was
    not found in classpath");
}
InputStreamReader fileReader = new InputStreamReader(fileStream, "ASCII");
```

3. Insert the XML data.
   a. Create a java.sql.preparedStatement object representing the SQL INSERT call. In the SQL INSERT statement, you must specify the name of the XML column to store the XML data. The column name must match the name that is defined in the Java metadata class.

      The following example shows how to create a preparedStatement object to insert data into the hospxml column in the HOSPITAL segment, using the java.sql.Connection instance conn.

      ```
      String s = "INSERT INTO pcb01.HOSPITAL (hospxml) VALUES (?)"
      PreparedStatement ps = conn.prepareStatement(s);
      ```

   b. Set the value of the XML data to insert into the preparedStatement object.

      The following table describes the methods and corresponding input data types that you can use to insert data in XML columns.

Table 96. Methods and data types for updating XML columns

Method	Input data type
PreparedStatement.setCharacterStream	Reader
PreparedStatement.setClob	Clob

In decomposed storage mode, XML data is stored with EBCDIC encoding. In intact storage mode, the default encoding is Unicode.

The following code sample shows how to insert XML data into the Hospital database.

```
package uxml.samples;

import java.io.*;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.PreparedStatement;
import com.ibm.ims.jdbc.IMSDataSource;

public class StoreXMLSamples{

    public static void main(String argv[]) throws SQLException,IOException {
        IMSDataSource ds = new IMSDataSource();
        ds.setMetadataURL("class://uxml.samples.BMP255NewSyntaxDatabaseView");
        ds.setDatastoreName("IMS1");
        ds.setDatastoreServer("yourhost.yourdomain.com");
        ds.setPortNumber(5555);
```

```
            ds.setDriverType(IMSDataSource.DRIVER_TYPE_4);
            ds.setUser("myUserID");
            ds.setPassword("myPass");

            // Specify file path of XML schema
            System.setProperty("http://www.ibm.com/ims/schema-resolver/file/path",
               "uxml/samples");

            Connection conn = null;

            try {
              conn = ds.getConnection();
              Statement st = conn.createStatement();
             String doc = "hospwashington.xml";
             StoreXMLSamples storeSample = new StoreXMLSamples();
             InputStream fileStream =
                 storeSample.getClass().getResourceAsStream(doc);
             if (fileStream == null) {
                 throw new FileNotFoundException("Insert Document: '" +
                     doc + "' was not found in classpath");
             }

              // Convert XML document from ASCII to Unicode
              InputStreamReader fileReader =
                 new InputStreamReader(fileStream, "ASCII");

             PreparedStatement ps =
                 conn.prepareStatement("INSERT INTO pcb01.HOSPITAL" +
                   " (hospxml) VALUES (?)");

           ps.setCharacterStream(1, fileReader, -1);
           int rows = ps.executeUpdate();
          System.out.println("Inserted");
         conn.commit();
               conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
                if (!conn.isClosed()) {
                    conn.rollback();
                    conn.close();
                }
            }
        }
}
```

Retrieving XML data by using the IMS Universal JDBC driver

You can use the IMS Universal JDBC driver to retrieve XML data from an IMS database as a character large object (CLOB) through an SQL SELECT statement.

To retrieve XML data in your IMS Universal JDBC driver application:

1. Specify the file path that contains the XML schema file (.xsd) describing the input XML data structure by setting the *http://www.ibm.com/ims/schema-resolver/file/path* environment variable. The following example shows how to programmatically set the environment variable. In this example, the file path `uxml/samples` indicates a relative path to the XML schema file. You can also specify an absolute file path.

   ```
   System.setProperty("http://www.ibm.com/ims/schema-resolver/file/path",
      "uxml/samples");
   ```

2. Specify and execute an SQL SELECT statement to retrieve the XML data. The database table in your SQL SELECT statement must include the XML column for retrieving the XML data. If you specify the column name explicitly in the SQL SELECT statement, the column name must match the name defined in the Java metadata class.

The following example shows how to obtain a java.sql.resultSet object from an SQL SELECT call to retrieve the hospxml column in the HOSPITAL segment. In the example, st is a java.sql.Statement instance.

```
ResultSet rs = st.executeQuery("SELECT hospxml FROM PCB01.HOSPITAL");
```

3. Read the XML data from the resultSet object after the retrieve call is made. The XML data is stored in a java.sql.Clob object in the resultSet.

The following code sample shows how to retrieve XML data from the Hospital database.

```
package uxml.samples;

import java.io.*;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import com.ibm.ims.jdbc.IMSDataSource;

public class RetrieveXMLSamples{

    public static void main(String argv[]) throws SQLException,IOException {
        IMSDataSource ds = new IMSDataSource();
        ds.setMetadataURL("class://uxml.samples.BMP255NewSyntaxDatabaseView");
        ds.setDatastoreName("IMS1");
        ds.setDatastoreServer("yourhost.yourdomain.com");
        ds.setPortNumber(5555);
        ds.setDriverType(IMSDataSource.DRIVER_TYPE_4);
        ds.setUser("myUserId");
        ds.setPassword("myPass");

        // Specify file path of XML schema
        System.setProperty("http://www.ibm.com/ims/schema-resolver/file/path",
           "uxml/samples");

        Connection conn = null;

        try {
            conn = ds.getConnection();

            Statement st = conn.createStatement();

        ResultSet rs = st.executeQuery("SELECT hospxml FROM PCB01.HOSPITAL");

        StringWriter sw = new StringWriter();

            while (rs.next()) {

                Clob clob = rs.getClob(1);
                Reader reader = clob.getCharacterStream();
                char[] buffer = new char[1000];
                int read = reader.read(buffer);
                while (read != -1) {
                 sw.write(buffer,0,read);
                    read = reader.read(buffer);
                }
            }

           String result = sw.toString();
           System.out.println(result);
           System.out.println();

           conn.commit();
           conn.close();
        } catch (SQLException e) {
```

```
                e.printStackTrace();
                if (!conn.isClosed()) {
                    conn.rollback();
                    conn.close();
                }
            }
        }
    }
```

Data transformation support for JDBC

The IMS JDBC drivers provide data transformation on behalf of client applications. When provided with the information from the Java database metadata class, the libraries are able to internally convert data from one datatype to another.

Supported JDBC data types

The following table lists the supported Java data types for each JDBC data type.

Table 97. Supported JDBC data types

JDBC data type	Java data type	Length
BIGINT	long	8 bytes
BINARY	byte[]	1 - 32 KB
BIT	Boolean	1 byte
CHAR	java.lang.String	1 - 32 KB
CLOB	java.sql.Clob	Application-defined
DATE	java.sql.Date	Application-defined
DOUBLE	double	8 bytes
FLOAT	float	4 bytes
INTEGER	int	4 bytes
PACKEDDECIMAL	java.math.BigDecimal	1 - 10 bytes
SMALLINT	short	2 bytes
TIME	java.sql.Time	Application-defined
TIMESTAMP	java.sql.Timestamp	Application-defined
TINYINT	byte	1 byte
ZONEDDECIMAL	java.math.BigDecimal	1 - 19 bytes

Methods for retrieving and converting data types

With the IMS classic JDBC driver and the IMS Universal JDBC driver, you can use the ResultSet interface (java.sql.ResultSet) to retrieve and convert the data from the type that is defined in the database metadata to the type that is required by your Java application. Similarly, with the IMS Universal DL/I driver, you can use the Path interface to perform data retrieval and conversion to Java data types.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

The following table shows the available get methods in the ResultSet interface (for the IMS classic JDBC driver and the IMS Universal JDBC driver) or the Path interface (for the IMS Universal DL/I driver) for accessing data of a certain Java data type.

The "No Truncation or Data Loss" column indicates the data types that are designed to be accessed with the given getXXX method. No truncation or data loss occurs when using those methods for those data types. The data types that are in the “Legal without Data Integrity” column are all other legal calls; however, data integrity cannot be ensured when using the given getxxx method to access those data types. If a data type is not in either column, using the given getXXX method for that data type will result in an exception.

Table 98. ResultSet.getXXX and Path.getXXX methods to retrieve data types

ResultSet.getXXX Method or Path.getXXX Method	**Data Type (any not listed result in an exception)**	
	No Truncation or Data Loss	**Legal without Data Integrity**
getByte	TINYINT UTINYINT	SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getShort	SMALLINT USMALLINT	TINYINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getInt	INTEGER UINTEGER	TINYINT SMALLINT BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getLong	BIGINT UBIGINT	TINYINT SMALLINT INTEGER FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]

Table 98. ResultSet.getXXX and Path.getXXX methods to retrieve data types (continued)

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getFloat	FLOAT	TINYINT SMALLINT INTEGER BIGINT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getDouble	DOUBLE	TINYINT SMALLINT INTEGER BIGINT FLOAT BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getBoolean	BIT	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getString	CHAR VARCHAR	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT PACKEDDECIMAL[1] ZONEDDECIMAL[1] BINARY DATE TIME TIMESTAMP
getBigDecimal	BINARY[3] PACKEDDECIMAL[1] ZONEDDECIMAL[1]	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR
getClob	CLOB[2]	all others result in an exception

Table 98. ResultSet.getXXX and Path.getXXX methods to retrieve data types (continued)

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getBytes	BINARY	all others result in an exception
getDate	DATE	CHAR VARCHAR TYIMESTAMP
getTime	TIME	CHAR VARCHAR TIMESTAMP
getTimestamp	TIMESTAMP	CHAR VARCHAR DATE TIME

Note:

1. PACKEDDECIMAL and ZONEDDECIMAL are data type extensions for the IMS classic JDBC driver , the IMS Universal JDBC driver, and the IMS Universal DL/I driver. All other types are standard SQL types defined in SQL92. **Restriction:** PACKEDDECIMAL and ZONEDDECIMAL data types do not support the Sign Leading or Sign Separate modes. For these two data types, sign information is always stored with the Sign Trailing method.
2. The CLOB data type is supported only for the retrieval and storage of XML data.
3. The BINARY data type is valid only for decimal data used with a binary type converter.

If the field type is either PACKEDDECIMAL or ZONEDDECIMAL, the type qualifier is the COBOL PICTURE string that represents the layout of the field. All COBOL PICTURE strings that contain valid combinations of 9s, Ps, Vs, and Ss are supported. Expansion of PICTURE strings is handled automatically. For example, '9(5)' is a valid PICTURE string. For zoned decimal numbers, the decimal point can also be used in the PICTURE string. `PIC 9(06)V99 COMP` and `PIC 9(06)V99 COMP-4` are valid PICTURE clauses for BINARY decimal data.

If the field contains DATE, TIME, or TIMESTAMP data, the type qualifier specifies the format of the data. For example, a type qualifier of *ddMMyyyy* indicates that the data is formatted as follows:

```
11122011 is December 11, 2011
```

For DATE and TIME types, all formatting options in the java.text.SimpleDateFormat class are supported.

For the TIMESTAMP type, the formatting option 'f' is available for nanoseconds. TIMESTAMP can contain up to nine 'f's and replaces the 'S' options for milliseconds. Instead, 'fff' indicates milliseconds of precision. An example TIMESTAMP format is as follows:

```
yyyy-mm-dd hh:mm:ss.fffffffff
```

COBOL copybook types that map to Java data types

Because data in IMS is not strongly typed, you can use COBOL copybook types to map your IMS data to Java data types.

The following table describes how COBOL copybook types are mapped to both DLITypeInfo constants in the DLIDatabaseView class and Java data types.

Table 99. Mapping from COBOL formats to DLITypeInfo constants and Java data types

Copybook format	DLITypeInfo constant	Java data type
PIC X	CHAR	`java.lang.String`
PIC 9 BINARY[1]	See "DLITypeInfo constants and Java data types based on the PICTURE clause".[2]	See "DLITypeInfo constants and Java data types based on the PICTURE clause".[2]
COMP-1	FLOAT	float
COMP-2	DOUBLE	double
PIC 9 COMP-3[3]	PACKEDDECIMAL	`java.math.BigDecimal`
PIC 9 DISPLAY[4]	ZONEDDECIMAL	`java.math.BigDecimal`

Notes:

1. Synonyms for BINARY data items are COMP and COMP-4. A `PIC 9(06)V99` statement with COMP or COMP-4 is used for binary decimal data.
2. For BINARY data items, the DLITypeInfo constant and Java type depend on the number of digits in the PICTURE clause.The table "DLITypeInfo constants and Java data types based on the PICTURE clause" describes the type based on PICTURE clause length.
3. PACKED-DECIMAL is a synonym for COMP-3.
4. If the USAGE clause is not specified at either the group or elementary level, it is assumed to be DISPLAY.

The following table shows the DLITypeInfo constants and the Java data types based on the PICTURE clause.

Table 100. DLITypeInfo constants and Java data types based on the PICTURE clause

Digits in PICTURE clause	Storage occupied	`DLITypeInfo` constant	Java data type
1 through 2	1 byte	TINYINT UTINYINT	byte
1 through 4	2 bytes	SMALLINT USMALLINT	short
5 through 9	4 bytes	INTEGER UINTEGER	int
10 through 18	8 bytes	BIGINT UBIGINT	long

The following table shows examples of specific copybook formats mapped to DLITypeInfo constants.

Table 101. Copybook formats mapped to DLITypeInfo constants

Copybook format	DLITypeInfo constant
PIC X(25)	CHAR
PIC 9(02) COMP	UTINYINT
PIC S9(04) COMP	SMALLINT
PIC 9(04) COMP	USMALLINT

Table 101. Copybook formats mapped to DLITypeInfo constants (continued)

Copybook format	DLITypeInfo constant
PIC S9(06) COMP-4	INTEGER
PIC 9(06) COMP-4	UINTEGER
PIC 9(06)V99 COMP or COMP-4	BINARY
PIC S9(12) BINARY	BIGINT
PIC 9(12) BINARY	UBIGINT
COMP-1	FLOAT
COMP-2	DOUBLE
PIC S9(06)V99	ZONEDDECIMAL
PIC 9(06).99	ZONEDDECIMAL
PIC S9(06)V99 COMP-3	PACKEDDECIMAL

Programming with the IMS Universal DL/I driver

Use the IMS Universal DL/I driver when you need to write granular queries to access IMS databases directly from a Java client in an non-managed environment.

Because of the fundamental differences between hierarchical databases and relational databases, sometimes the JDBC API does not provide access to the full set of IMS databases features. The IMS Universal DL/I driver is closely related to the traditional IMS DL/I database call interface that is used with other programming languages for writing applications in IMS, and provides a lower-level access to IMS database functions than the JDBC API. By using the IMS Universal DL/I driver, you can build segment search arguments (SSAs) and use the methods of the program communication block (PCB) object to read, insert, update, delete, or perform batch operations on segments. You can gain full navigation control in the segment hierarchy.

Preparing to write a Java application with the IMS Universal drivers

Java application programs that use the IMS Universal drivers require the Java Development Kit 5.0 (JDK 5.0). Java programs that run in JMP and JBP regions require the Java Development Kit 6.0 (JDK 6.0) or later. Java application programs that use the IMS Universal drivers also require a way to generate the IMS database metadata, such as using the IMS Enterprise Suite Explorer for Development. The default segment encoding of the database metadata class produced by the IMS Explorer for Development is `cp1047`. To change the segment encoding, use the com.ibm.ims.base.DLIBaseSegment.setDefaultEncoding method.

Basic steps in writing a IMS Universal DL/I driver application

In general, to write a application program with the IMS Universal DL/I driver, you need to complete the following tasks.

To write an IMS Universal DL/I driver application, follow these steps.

1. Import the `com.ibm.ims.dli` package that contains the IMS Universal DL/I driver classes, interfaces, and methods.
2. Connect to an IMS database subsystem.
3. Obtain a program specification block (PSB), which contains one or more PCBs.

4. Obtain a PCB handle, which defines an application's view of an IMS database and provides the ability to issue database calls to retrieve, insert, update, and delete database information.
5. Obtain an unqualified segment search argument list (SSAList) of one or more segments in the database hierarchy.
6. Add qualification statements to specify the segments targeted by DL/I calls.
7. If retrieving data, mark the segment fields to be returned.
8. Execute DL/I calls to the IMS database.
9. Handle errors that are returned from the DL/I programming interface.
10. Disconnect from the IMS database subsystem.

Related tasks:

"Retrieving data in a IMS Universal DL/I driver application" on page 622

Related reference:

"Generating the runtime Java metadata class" on page 557

Java packages for IMS Universal DL/I driver support

Before you can invoke IMS Universal DL/I driver methods, you must access all or parts of various Java packages that contain those methods.

You can do that by either importing the packages or specific classes, or by using the fully-qualified class names. You might need the following packages or classes for your IMS Universal DL/I driver application:

com.ibm.ims.dli
Contains the core classes, interfaces, and methods for the IMS Universal DL/I driver.

com.ibm.ims.base
Contains exception classes for errors that are returned by DL/I or IMS.

Related reference:

Java API documentation (Javadoc) (Application Programming APIs)

Connecting to an IMS database by using the IMS Universal DL/I driver

Before you can execute DL/I calls from your IMS Universal DL/I driver application, you must connect to an IMS database.

The IMS Universal DL/I driver application can establish a connection to an IMS database using the PSB interface, which is part of the `com.ibm.ims.dli` package. Pass the connection properties using an IMSConnectionSpec instance.

To connect to an IMS database by using the IMS Universal DL/I driver:

1. Create an IMSConnectionSpec instance by calling the createIMSConnectionSpec method in the IMSConnectionSpecFactory class.
2. Set the following connection properties for the IMSConnectionSpec instance.

   DatastoreName

   The name of the IMS data store to access.

   - When using type-4 connectivity, the **`DatastoreName`** property must match either the name of the data store defined to ODBM or be blank. The data store name is defined in the ODBM CSLDCxxx PROCLIB member using either the DATASTORE(NAME=*name*) or

DATASTORE(NAME=*name*, ALIAS(NAME=*aliasname*)) parameter. If an alias is specified, you must specify the *aliasname* as the value of the **datastoreName** property. If the **DatastoreName** value is left blank (or not supplied), IMS Connect connects to any available instance of ODBM as it is assumed that data sharing is enabled between all datastores defined to ODBM.

- When using type-2 connectivity, set the **DatastoreName** property to the IMS subsystem alias. This is not required to be set for the Java Dependent Region run time.

MetadataURL

The location of the database metadata representing the target IMS database.

The **MetadataURL** property is the fully qualified name of the Java metadata class generated by the IMS Enterprise Suite Explorer for Development. The URL must be prefixed with `class://` (for example, `class://com.foo.BMP255DatabaseView`).

In a J2C Connection Factory environment, the **MetadataURL** property can be overridden for an individual connection without affecting the default value specified for the resource adapter.

PortNumber

The TCP/IP server port number to be used to communicate with IMS Connect. The port number is defined using the DRDAPORT parameter on the ODACCESS statement in the IMS Connect configuration PROCLIB member. The default port number is 8888. Do not set this property when using type-2 connectivity.

DatastoreServer

The name or IP address of the data store server (IMS Connect). You can provide either the host name (for example, dev123.svl.ibm.com) or the IP address (for example, 192.166.0.2). Do not set this property when using type-2 connectivity.

DriverType

The type of driver connectivity to use (value must be `IMSConnectionSpec.DRIVER_TYPE_4` for type-4 connectivity or `IMSConnectionSpec.DRIVER_TYPE_2` for type-2 connectivity).

sslConnection

Optional. Indicates if this connection uses Secure Sockets Layer (SSL) for data encryption. Set this property to "true" to enable SSL, or to "false" otherwise. Do not set this property when using type-2 connectivity.

loginTimeout

Optional. Specifies the number of seconds that the driver waits for a response from the server before timing out a connection initialization or server request. Set this property to a non-negative integer for the number of seconds. Set this property to 0 for an infinite timeout length. Do not set this property when using type-2 connectivity.

user The user name for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

password
The password for the connection to IMS Connect provided by your RACF administrator. Do not set this property when using type-2 connectivity.

dbViewLocation
Optional. Specifies the fully qualified path to a databaseView metadata class. You can use this property to include a metadata class that is not located in your project path.

3. Pass the connection request properties to the PSBFactory class to create the PSB instance. When the PSB instance is created successfully, a connection is established to the database.
4. When you are finished with a connection to the IMS database from a IMS Universal DL/I driver application, you must close the connection to the database by calling the close method on the PSB instance.

Example: type-4 Connection

The following code example shows how to create a type-4 connection to an IMS database from your IMS Universal DL/I driver application:

```
IMSConnectionSpec connSpec = IMSConnectionSpecFactory.createIMSConnectionSpec();
connSpec.setDatastoreName("SYS1");
connSpec.setDatastoreServer("9.876.543.21");
connSpec.setPortNumber(8888);
connSpec.setMetadataURL("class://testdb.jdbo.HospitalDatabaseView");
connSpec.setSSLConnection(true);
connSpec.setLoginTimeout(10);
connSpec.setUser("usr");
connSpec.setPassword("usrpwd");
connSpec.setDriverType(IMSConnectionSpec.DRIVER_TYPE_4);
PSB psb = PSBFactory.createPSB(connSpec);
```

Related tasks:

"Configuring the IMS Universal drivers for SSL support" on page 637

Related reference:

Java API documentation (Javadoc) (Application Programming APIs)

IMS Universal DL/I driver interfaces for executing DL/I operations

In a traditional IMS application, you make DL/I calls to insert, update, delete, or retrieve data. To perform the same functions in a IMS Universal DL/I driver application, you invoke methods.

Methods are defined in the following interfaces:

- The program specification block (PSB) interface is used to connect to IMS databases. Use the PSB interface to obtain a handle to any program communication block (PCB) that is contained in the PSB. The PCB handle is used to access the particular database that is referenced by the PCB.
- The PCB interface represents a cursor position in an IMS database. The PCB interface supports DL/I message call functions, including Get Unique (GU), Get Next (GN), Get Next Within Parent (GNP), Insert (ISRT), Replace (REPL), and Delete (DLET). The PCB interface can obtain an unqualified list of segment search arguments and perform batch retrieve, update, and delete operations. You can also use the PCB interface to return the application interface block (AIB) that is associated with the most recent DL/I call.

- The SSAList interface represents a list of segment search arguments (SSAs) used to specify the segments to target in a particular database call. Use the SSAList interface to construct the SSAs, and to set the command codes and lock class for the SSAs. You can set an initial qualification statement and append additional qualifiers, based on the values of the segment fields, to restrict which segments to target in the DL/I call. You can also specify which fields to return from a database retrieve call.
- The Path interface represents a database record for the purpose of a DL/I retrieval or update operation. The Path interface can be viewed as the concatenation of all of the segment instances in a specific database hierarchic path, starting from the highest level segment that is nearest the root segment to the lowest level segment. Use the Path interface to set or retrieve the value of any segment field that is located in the hierarchic path.
- The PathSet interface provides access to a collection of Path objects that are returned by a batch retrieve operation.
- The AIB interface and the database PCB (DBPCB) interface return useful information that was returned by IMS as a result of a DL/I call.
- The GSAMPCB interface represents a GSAM PCB and is essentially a cursor position in a GSAM database. This interface provides data access to GSAM databases with calls that are similar to DL/I calls.
- The RSA interface represents a GSAM database record search argument that is the key to a cursor position in the GSAM database.

Related reference:

Java API documentation (Javadoc) (Application Programming APIs)

Specifying segment search arguments using the SSAList interface

The SSAList interface represents a set of a list of segment search arguments used to specify the segments to target in a particular database call.

Use the SSAList interface to construct each segment search argument (SSA) in the list and to set the command codes and lock class for the SSAs. Each SSA in the SSAList can be unqualified or qualified.

In addition, your application can specify which segment fields are to be returned from a database retrieve call by using the markFieldForRetrieval or the markAllFieldsForRetrieval methods. Following the IMS default, all of the fields in the lowest level segment specified by the SSAList are initially marked for retrieval.

- For non-batch DL/I data retrieval or update operations, use the getPathForRetrieveReplace method.
- For a DL/I insert call, use the getPathForInsert method.
- For a batch update operation, use the getPathForBatchUpdate method.

The following examples demonstrate how to specify segment search arguments using the SSAList interface. The examples are based on the Hospital database.

Creating an unqualified SSAList

This example returns a Path that consists of all fields in the segment “DOCTOR”:

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","DOCTOR");
Path path = ssaList.getPathForRetrieveReplace();
pcb.getUnique(path, ssaList, false);
```

In the previous example, the ssaList represents all segments along the hierarchic path from the topmost segment ("HOSPITAL") to the lowest segment ("DOCTOR"). The ssaList will look like this:

```
HOSPITALb
WARDbbbbb
PATIENTbb
ILLNESSbb
TREATMNTb
DOCTORbbb
```

Creating a qualified SSAList

An SSAList can be qualified to filter the segments on the hierarchic path to be retrieved or updated. The general steps to create a qualified SSAList are:

1. Obtain an unqualified SSAList from the PCB using the getSSAList method.
2. Use the addInititalQualification method to specify the initial search criteria for a segment on the SSAList returned from a getSSAList method. For each segment represented in the SSAList, you can make one call to specify an initial qualification for that segment. The segment can be referenced by name or by using the 1-based offset of the SSA representing that segment within the SSAList. If you use more than one addInitialQualification statement for a segment, an exception will be thrown. The relational operator (**relationalOp**) parameter in the addInitialQualification method indicates the conditional criteria that the segment must meet in order to be qualified. Valid relational operators are:
   - EQUALS
   - GREATER_OR_EQUAL
   - GREATER_THAN
   - LESS_OR_EQUAL
   - LESS_THAN
   - NOT_EQUAL
3. To specify additional search criteria, use the appendQualification method. For each segment, you can make multiple calls to the appendQualification method to add more than one qualification statement. The Boolean operator (**booleanOp**) parameter in the appendQualification method indicates how this qualification is logically connected to the previous qualification. Valid Boolean operators are:
   - AND
   - OR
   - INDEPENDENT_AND
4. You can also qualify a SSAList by setting DL/I command codes and lock classes. The supported DL/I command codes include:
   - CC_A: The A command code (clear positioning).
   - CC_C: The C command code (concatenated key). Use the addConcatenatedKey method to add a concatenated key to a segment.
   - CC_D: The D command code (path call)
   - CC_F: The F command code (first occurrence)
   - CC_G: The G command code (prevent randomization).
   - CC_L: The L command code (last occurrence)
   - CC_N: The N command code (path call ignore)
   - CC_O: The O command code (contain field names or segment position and length).

- CC_P: The P command code (set parentage)
- CC_U: The U command code (maintain position at this level)
- CC_V: The V command code (maintain position at this level and all superior levels)

You can use a lock class to prevent another program from updating a segment until your program reaches a commit point. Use the addLockClass method to add a lock class to a segment. The supported lock class letters are “A” to “J”. The behavior of a lock class is the same as using a “Q” command code with that lock class letter.

The following code example demonstrates how to specify and use a qualified SSAList with a single initial qualification statement to retrieve data:

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","DOCTOR");
ssaList.addInitialQualification("PATIENT","PATNAME",SSAList.EQUALS,"ANDREA SMITH");
ssaList.markFieldForRetrieval("ILLNESS","ILLNAME",true);
ssaList.markFieldForRetrieval("TREATMNT","TREATMNT",true);
Path path = ssaList.getPathForRetrieveReplace();
pcb.getUnique(path, ssaList, false);
```

For the code example above, the ssaList will look like this:

```
HOSPITALb
WARDbbbbb
PATIENTb(PATNAMEbEQANDREAbSMITHbbbbb)
ILLNESSb*D
TREATMNT*D
```

The code example above retrieves the following information for all records where PATNAME is "ANDREA SMITH":

- The ILLNAME field in the ILLNESS segment.
- The TREATMNT field in the TREATMNT segment.
- All fields in the DOCTOR segment (by default, IMS returns all fields in the lowest level segment specified by the SSAList).

The following code example demonstrates how to specify a qualified SSAList with a multiple qualification statements to retrieve data:

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","WARD");
ssaList.addInitialQualification("WARD","NURCOUNT",SSAList.GREATER_THAN,4);
ssaList.appendQualification("WARD",SSAList.AND,"DOCCOUNT",SSAList.GREATER_THAN, 2);
```

For the code example above, the ssaList will look like this:

```
HOSPITALb
WARDbbbb(NURCOUNTGT4&DOCCOUNTGT2;)
```

The following example shows how to specify a qualified SSAList with the command code CC_L (which means "last occurence") to find the most recently admitted patient in the "SANTA TERESA" hospital:

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","PATIENT");
ssaList.addInitialQualification
   ("HOSPITAL","HOSPNAME",SSAList.EQUALS,"SANTA TERESA");
ssaList.addCommandCode("PATIENT",SSAList.CC_L);
Path path = ssaList.getPathForRetrieveReplace();
pcb.getUnique(path,ssaList,false);
```

For the code example above, the ssaList will look like this:

```
HOSPITAL(HOSPNAMEEQSANTAbTERESAbbbbb)
WARDbbbbb
PATIENTb*L
```

Debugging an SSAList

If you need to debug by identifying whether the segment search arguments in your SSAList are correct, use the buildSSAListInBytes method to build the SSAList in DL/I format for further debugging:

```
byte[][] ssaListInBytes = ssaList.buildSSAListInBytes();
```

You can iterate over each segment search argument in the byte array that is returned and print it out to make sure the segment search argument is what it should be.

Related concepts:

"Segment search arguments (SSAs)" on page 182

Related reference:

"SSA coding formats" on page 241

"Hospital database example" on page 557

Command code reference (Application Programming APIs)

Retrieving data in a IMS Universal DL/I driver application

The IMS Universal DL/I driver provides support for data retrieval that mirrors DL/I semantics.

The following are the general steps to retrieve segments from the database:

1. Obtain an SSAList instance from the PCB instance representing the database.
2. Optionally, you can add qualification statements to the SSAList instance.
3. Specify the segment fields to retrieve. Use the markFieldForRetrieval method to mark a single field, or use the markAllFieldsForRetrieval method to mark all the fields for a segment. Following the IMS default, all of the fields in the lowest-level segment specified by the SSAList instance are initially marked for retrieval. When one or more of the fields in the lowest level segment specified in an SSAList instance is marked for retrieval, only the explicitly marked fields are retrieved.
4. Get a Path instance by using the SSAList instance from the previous steps and calling the getPathForRetrieveReplace method. When a retrieve call is made, the resulting Path object will contain all the fields that have been marked for retrieval.
5. Call a DL/I retrieve operation using one of the following methods from the PCB interface:

Java API for DL/I retrieve method	Usage
getUnique	Retrieves a specific unique segment. This method provides the same functionality as the DL/I Get Unique (GU) database call. If the isHoldCall parameter is set to true, the call behaves as a DL/I Get Hold Unique (GHU) database call.

Java API for DL/I retrieve method	Usage
getNext	Retrieves the next segment in a Path. This method provides the same functionality as the DL/I Get Next (GN) database call. If the isHoldCall parameter is set to true, the call behaves as a DL/I Get Hold Next (GHN) database call.
getNextWithinParent	Retrieves the next segment within the same parent. This method provides the same functionality as the DL/I Get Next Within Parent (GNP) database call. If the isHoldCall parameter is set to true, the call behaves as a DL/I Get Hold Next Within Parent (GHNP) database call.
batchRetrieve	Retrieves multiple segments with a single call. See "Batch data retrieval in a Java API for DL/I application" for more information about how to use this method.

6. Read the values of the retrieved fields out of the Path object after the retrieve call is made.

IMS Universal DL/I driver data retrieval example

The following code fragment illustrates how to use the getUnique method and getNext method to retrieve the hospital name (HOSPNAME), ward name (WARDNAME), patient count (PATCOUNT), nurse count (NURCOUNT), and doctor count (DOCCOUNT) fields from the Hospital database:

```
import com.ibm.ims.dli.*;


public class HospitalDLIReadClient {

   public static void main(String[] args) {
     PSB psb = null;
     PCB pcb = null;
     SSAList ssaList = null;
     Path path = null;
     PathSet pathSet = null;

     try {
        // establish a database connection
        IMSConnectionSpec connSpec
           = IMSConnectionSpecFactory.createIMSConnectionSpec();
        connSpec.setDatastoreName("IMS1");
        connSpec.setDatastoreServer("ecdev123.svl.ibm.com");
        connSpec.setPortNumber(5555);
        connSpec.setMetadataURL("class://BMP266.BMP266DatabaseView");
        connSpec.setUser("usr");
        connSpec.setPassword("password");
    connSpec.setDriverType(IMSConnectionSpec.DRIVER_TYPE_4);
        psb = PSBFactory.createPSB(connSpec);
        System.out.println("**** Created a connection to the IMS database");

        pcb = psb.getPCB("PCb01");
        System.out.println("**** Created PCB object");

        // specify the segment search arguments
        ssaList = pcb.getSSAList("HOSPITAL", "WARD");
        // add the initial qualification
        ssaList.addInitialQualification("HOSPITAL", "HOSPCODE",
```

```
                    SSAList.GREATER_OR_EQUAL, 444);
                 // specify the fields to retrieve
                 ssaList.markFieldForRetrieval("HOSPITAL", "HOSPNAME", true);
                 ssaList.markAllFieldsForRetrieval("WARD", true);
                 ssaList.markFieldForRetrieval("WARD", "WARDNO", false);
                 System.out.println("**** Created SSAList object");

                 // obtain a Path containing the segments that match the SSAList criteria
                 path = ssaList.getPathForRetrieveReplace();
                 System.out.println("**** Created Path object");

                 // issue a DL/I GU call to retrieve the first segment on the Path
                if (pcb.getUnique(path, ssaList, true) {
                   System.out.println("HOSPNAME: "+ path.getString("HOSPITAL", "HOSPNAME"));
                   System.out.println("WARDNAME: "+ path.getString("WARD", "WARDNAME"));
                   System.out.println("PATCOUNT: "+ path.getInt("WARD", "PATCOUNT"));
                   System.out.println("NURCOUNT: "+ path.getInt("WARD", "NURCOUNT"));
                   System.out.println("DOCCOUNT: "+ path.getShort("WARD", "DOCCOUNT"));
                }

                // issue multiple DL/I GN calls until there are no more segments to retrieve
                while (pcb.getNext(pat, ssaList, true) {
                   System.out.println("HOSPNAME: "+ path.getString("HOSPITAL", "HOSPNAME"));
                   System.out.println("WARDNAME: "+ path.getString("WARD", "WARDNAME"));
                   System.out.println("PATCOUNT: "+ path.getInt("WARD", "PATCOUNT"));
                   System.out.println("NURCOUNT: "+ path.getInt("WARD", "NURCOUNT"));
                   System.out.println("DOCCOUNT: "+ path.getShort("WARD", "DOCCOUNT"));
                }

                // close the database connection
                psb.close();
                System.out.println("**** Disconnected from IMS database");

             } catch (DLIException e) {
                System.out.println(e);
                System.exit(0);
             }
          }
       }
```

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Related tasks:

"Basic steps in writing a IMS Universal DL/I driver application" on page 615

"Batch data retrieval in a IMS Universal DL/I driver application"

Related reference:

"Methods for retrieving and converting data types" on page 610

Batch data retrieval in a IMS Universal DL/I driver application

Use the batchRetrieve method to retrieve multiple segments in a single call.

Instead of the client application making multiple GU and GN calls, IMS will perform all of the GU and GN processing and will deliver the results back to the client in a single batch network operation. The fetch size property determines how much data is sent back on each batch network operation.

To perform a batch data retrieval operation:

1. Obtain an SSAList instance from the PCB instance that represents the database.
2. Optionally, you can add qualification statements to the SSAList instance.
3. Specify the segment fields to retrieve. Use the markFieldForRetrieval method to mark a single field, or use the markAllFieldsForRetrieval method to mark all

the fields for a segment. Following the IMS default, all of the fields in the lowest-level segment specified by the SSAList instance are initially marked for retrieval.

4. Optionally, set the fetch size property. The fetch size gives a hint to the IMS Universal DL/I driver as to the number of records to fetch from the database in a single batch operation. See "Improving query performance by setting fetch size" for more information.
5. Call the batchRetrieve method with the SSAList instance above as a parameter. The batchRetrieve method returns a PathSet that contains a list of records that satisfy the criteria specified by the SSAList.
6. Read the values of the retrieved fields out of the Path object after the retrieve call is made.

IMS Universal DL/I driver batch data retrieval example

The following code fragment illustrates how to use the batchRetrieve method to retrieve the hospital name (HOSPNAME), ward name (WARDNAME), patient count (PATCOUNT), nurse count (NURCOUNT), and doctor count (DOCCOUNT) fields from the Hospital database:

```
import com.ibm.ims.dli.*;


public class HospitalDLIReadClient {

   public static void main(String[] args) {
     PSB psb = null;
     PCB pcb = null;
     SSAList ssaList = null;
     Path path = null;
     PathSet pathSet = null;

     try {
        // establish a database connection
        IMSConnectionSpec connSpec
           = IMSConnectionSpecFactory.createIMSConnectionSpec();
        connSpec.setDatastoreName("IMS1");
        connSpec.setDatastoreServer("ecdev123.svl.ibm.com");
        connSpec.setPortNumber(5555);
        connSpec.setMetadataURL("class://BMP266.BMP266DatabaseView");
        connSpec.setUser("usr");
        connSpec.setPassword("password");
    connSpec.setDriverType(IMSConnectionSpec.DRIVER_TYPE_4);

        psb = PSBFactory.createPSB(connSpec);
        System.out.println("**** Created a connection to the IMS database");

        pcb = psb.getPCB("PCb01");
        System.out.println("**** Created PCB object");

        // specify the segment search arguments
        ssaList = pcb.getSSAList("HOSPITAL", "WARD");
        // add the initial qualification
        ssaList.addInitialQualification("HOSPITAL", "HOSPCODE",
           SSAList.GREATER_OR_EQUAL, 444);
        // specify the fields to retrieve
        ssaList.markFieldForRetrieval("HOSPITAL", "HOSPNAME", true);
        ssaList.markAllFieldsForRetrieval("WARD", true);
        ssaList.markFieldForRetrieval("WARD", "WARDNO", false);
        System.out.println("**** Created SSAList object");

        // issue the database call to perform a batch retrieve operation
        pathSet = pcb.batchRetrieve(ssaList);
```

```
            System.out.println("**** Batch Retrieve returned without exception");
            System.out.println("**** Created PathSet object");

            while(pathSet.hasNext()){
               path = pathSet.next();

               System.out.println("HOSPNAME: "+ path.getString("HOSPITAL", "HOSPNAME"));
               System.out.println("WARDNAME: "+ path.getString("WARD", "WARDNAME"));
               System.out.println("PATCOUNT: "+ path.getInt("WARD", "PATCOUNT"));
               System.out.println("NURCOUNT: "+ path.getInt("WARD", "NURCOUNT"));
               System.out.println("DOCCOUNT: "+ path.getShort("WARD", "DOCCOUNT"));
            }
            System.out.println("**** Fetched all rows from PathSet");

            // close the database connection
            psb.close();
            System.out.println("**** Disconnected from IMS database");
         } catch (DLIException e) {
            System.out.println(e);
            System.exit(0);
         }
      }
}
```

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Related reference:

"Methods for retrieving and converting data types" on page 610

Improving query performance by setting fetch size:

You can optimize query performance by setting the number of records to retrieve in batch retrieval mode.

In the IMS Universal DL/I driver, a list of rows is represented by a Path instance containing one or more segments that match the segment search argument criteria specified by an SSAList. The *fetch size* is the number of rows physically retrieved from the IMS database per network call. This is set for you internally. You can also set the fetch size using the setFetchSize method from the PCB interface. Setting the fetch size allows a single request to return multiple rows at a time, so that each application request to retrieve the next row does not always result in a network request. If the fetch size was *n* and the IMS Universal DL/I driver application requires more than the previous *n* number of rows during a batch retrieve operation, another network call will be made on behalf of the application to retrieve the next *n* number of rows that match the segment search argument criteria.

Methods for retrieving and converting data types

With the IMS classic JDBC driver and the IMS Universal JDBC driver, you can use the ResultSet interface (java.sql.ResultSet) to retrieve and convert the data from the type that is defined in the database metadata to the type that is required by your Java application. Similarly, with the IMS Universal DL/I driver, you can use the Path interface to perform data retrieval and conversion to Java data types.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

The following table shows the available get methods in the ResultSet interface (for the IMS classic JDBC driver and the IMS Universal JDBC driver) or the Path interface (for the IMS Universal DL/I driver) for accessing data of a certain Java data type.

The "No Truncation or Data Loss" column indicates the data types that are designed to be accessed with the given getXXX method. No truncation or data loss occurs when using those methods for those data types. The data types that are in the “Legal without Data Integrity” column are all other legal calls; however, data integrity cannot be ensured when using the given getxxx method to access those data types. If a data type is not in either column, using the given getXXX method for that data type will result in an exception.

Table 102. ResultSet.getXXX and Path.getXXX methods to retrieve data types

ResultSet.getXXX Method or Path.getXXX Method	**Data Type (any not listed result in an exception)**	
	No Truncation or Data Loss	**Legal without Data Integrity**
getByte	TINYINT UTINYINT	SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getShort	SMALLINT USMALLINT	TINYINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getInt	INTEGER UINTEGER	TINYINT SMALLINT BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getLong	BIGINT UBIGINT	TINYINT SMALLINT INTEGER FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]

Table 102. ResultSet.getXXX and Path.getXXX methods to retrieve data types (continued)

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getFloat	FLOAT	TINYINT SMALLINT INTEGER BIGINT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getDouble	DOUBLE	TINYINT SMALLINT INTEGER BIGINT FLOAT BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getBoolean	BIT	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getString	CHAR VARCHAR	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT PACKEDDECIMAL[1] ZONEDDECIMAL[1] BINARY DATE TIME TIMESTAMP
getBigDecimal	BINARY[3] PACKEDDECIMAL[1] ZONEDDECIMAL[1]	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR
getClob	CLOB[2]	all others result in an exception

Table 102. ResultSet.getXXX and Path.getXXX methods to retrieve data types (continued)

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getBytes	BINARY	all others result in an exception
getDate	DATE	CHAR VARCHAR TYIMESTAMP
getTime	TIME	CHAR VARCHAR TIMESTAMP
getTimestamp	TIMESTAMP	CHAR VARCHAR DATE TIME

Note:

1. PACKEDDECIMAL and ZONEDDECIMAL are data type extensions for the IMS classic JDBC driver , the IMS Universal JDBC driver, and the IMS Universal DL/I driver. All other types are standard SQL types defined in SQL92. **Restriction:** PACKEDDECIMAL and ZONEDDECIMAL data types do not support the Sign Leading or Sign Separate modes. For these two data types, sign information is always stored with the Sign Trailing method.
2. The CLOB data type is supported only for the retrieval and storage of XML data.
3. The BINARY data type is valid only for decimal data used with a binary type converter.

If the field type is either PACKEDDECIMAL or ZONEDDECIMAL, the type qualifier is the COBOL PICTURE string that represents the layout of the field. All COBOL PICTURE strings that contain valid combinations of 9s, Ps, Vs, and Ss are supported. Expansion of PICTURE strings is handled automatically. For example, '9(5)' is a valid PICTURE string. For zoned decimal numbers, the decimal point can also be used in the PICTURE string. `PIC 9(06)V99 COMP` and `PIC 9(06)V99 COMP-4` are valid PICTURE clauses for BINARY decimal data.

If the field contains DATE, TIME, or TIMESTAMP data, the type qualifier specifies the format of the data. For example, a type qualifier of *ddMMyyyy* indicates that the data is formatted as follows:

```
11122011 is December 11, 2011
```

For DATE and TIME types, all formatting options in the java.text.SimpleDateFormat class are supported.

For the TIMESTAMP type, the formatting option 'f' is available for nanoseconds. TIMESTAMP can contain up to nine 'f's and replaces the 'S' options for milliseconds. Instead, 'fff' indicates milliseconds of precision. An example TIMESTAMP format is as follows:

```
yyyy-mm-dd hh:mm:ss.fffffffff
```

Creating and inserting data in a IMS Universal DL/I driver application

Use the create or the insert methods in the PCB interface to add a new segment to the database.

In the IMS Universal DL/I driver, the insert and create methods provide functionality similar to the DL/I ISRT call. The insert methods will return an IMS status code indicating the results of the DL/I operation, whereas the create method returns the number of segments created (this will always return 1). An exception is thrown if a key field is not set.

The following are the general steps to add a new segment to the database:

1. Obtain an SSAList instance from the PCB instance representing the database.
2. Optionally, you can add qualification statements to the SSAList.
3. Get a Path instance by using the SSAList instance from the previous steps and calling the getPathForInsert method. The getPathForInsert method takes the name of an existing segment on the SSAList as a parameter. The parameter indicates the name of the segment type for the new segment. For instance, to add a new patient segment, you would pass the segment name PATIENT as the parameter.
4. Using the Path instance from the step above, set the field values for the new segment.
5. Call the insert or create method to add the new segment.

IMS Universal DL/I driver create and insert example

The following code fragment illustrates how to use the create and insert methods to add a new patient and illness segment in the database where the hospital name is “SANTA TERESA” and the ward name is “GENERAL”.

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","PATIENT");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",
SSAList.EQUALS,"SANTA TERESA");
ssaList.addInitialQualification("WARD","WARDNAME",
SSAList.EQUALS,"GENERAL");

Path path = ssaList.getPathForInsert("PATIENT");
path.setString("PATIENT", "PATNUM", "0088");
path.setString("PATIENT", "PATNAME", "JACK KIRBY");
int i = pcb.create(path, ssaList); // returns i = 1 if successful
System.out.println(i);

SSAList ssaList2 = pcb.getSSAList("HOSPITAL","ILLNESS");
ssaList2.addInitialQualification("HOSPITAL","HOSPNAME",
   SSAList.EQUALS,"SANTA TERESA");
ssaList2.addInitialQualification("WARD","WARDNAME",
   SSAList.EQUALS,"GENERAL");
ssaList2.addInitialQualification("PATIENT","PATNUM",
   SSAList.EQUALS,"0088");

Path path2 = ssaList2.getPathForInsert("ILLNESS");
path2.setString("ILLNAME", "APPENDICITIS");
short status = pcb.insert(path2, ssaList2);
```

The following code example shows another way to do this in a single call:

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","ILLNESS");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",
   SSAList.EQUALS,"SANTA TERESA");
ssaList.addInitialQualification("WARD","WARDNAME",
   SSAList.EQUALS,"GENERAL");
```

```
ssaList.addCommandCode("PATIENT", SSAList.CC_D);

Path path = ssaList.getPathForInsert("PATIENT");
path.setString("PATIENT", "PATNUM", "0088");
path.setString("PATIENT", "PATNAME", "JACK KIRBY");
path.setString("ILLNAME", "APPENDICITIS");

int i = pcb.create(path, ssaList); // returns i = 1 if successful
```

Important: To persist changes made to the database, your application must call the PSB.commit method prior to deallocating the PSB, otherwise the changes are rolled back up to the last point commit was called.

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Updating data in a IMS Universal DL/I driver application

Use the replace methods in the PCB interface to update an existing segment in the database.

In the IMS Universal DL/I driver, the replace methods provide functionality similar to the DL/I REPL call. The replace methods will return an IMS status code indicating the results of the DL/I operation.

The following are the general steps to update an existing segment in the database:

1. Obtain an SSAList instance from the PCB instance representing the database.
2. Optionally, you can add qualification statements to the SSAList instance. See "Specifying segment search arguments using the SSAList interface" for more information.
3. Get a Path instance by using the SSAList instance from the previous steps and calling the getPathForRetrieveReplace method.
4. Using the Path instance from the step above, set the field values to update for the segment.
5. Perform a Hold operation before issuing the replace call. The Hold operation can be a getUnique, getNext, or getNextWithinParent method call.
6. Call the replace method to update the segment.

IMS Universal DL/I driver update example

The following code fragment illustrates how to use the replace method to update a patient's name in patient records where the patient name is "ANDREA SMITH", the ward name is "SURG", and the hospital name is "ALEXANDRIA".

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","PATIENT");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",SSAList.EQUALS,"ALEXANDRIA");
ssaList.addInitialQualification("WARD","WARDNAME",SSAList.EQUALS,"SURG");
ssaList.addInitialQualification("PATIENT","PATNAME",SSAList.EQUALS,"ANDREA SMITH");

Path path = ssaList.getPathForRetrieveReplace();
if(pcb.getUnique(path, ssaList, true)){
   path.setString("PATNAME", "ANDREA TAYLOR");
   pcb.replace(path);
}
while(pcb.getNext(path, ssaList, true){
   path.setString("PATNAME", "ANDREA TAYLOR");
   pcb.replace(path);
}
```

Note: To persist changes made to the database, your application must call the commit method prior to deallocating the PSB, otherwise the changes are rolled back up to the last point the commit method was called.

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Making batch data updates in IMS Universal DL/I driver applications

Use the batchUpdate method in the PCB interface to update multiple existing segments in the database with one call.

The following are the general steps to update multiple existing segments in the database with a single call:

1. Obtain an SSAList instance from the PCB instance representing the database.
2. Optionally, you can add qualification statements to the SSAList method. See "Specifying segment search arguments using the SSAList interface" for more information.
3. Get a Path instance by using the SSAList instance from the previous steps and calling the getPathForBatchUpdate method.
4. Using the Path instance from the step above, set the field values to update for the segments.
5. Call the batchUpdate method to update the segments.

The following code fragment illustrates how to use the batchUpdate method to modify a patient's name. The SSAList instance is set to update only records where the patient name is "ANDREA SMITH", the ward name is "SURG", and the hospital name is "ALEXANDRIA". The getPathForBatchUpdate method is called to obtain a Path containing the PATIENT segment and its child segments. Finally, the batchUpdate method is called to change the value of the patient name field to "ANDREA TAYLOR".

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","PATIENT");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",SSAList.EQUALS,"ALEXANDRIA");
ssaList.addInitialQualification("WARD","WARDNAME",SSAList.EQUALS,"SURG");
ssaList.addInitialQualification("PATIENT","PATNAME",SSAList.EQUALS,"ANDREA SMITH");
Path path = ssaList.getPathForBatchUpdate("PATIENT");
path.setString("PATNAME", "ANDREA TAYLOR");
pcb.batchUpdate(path, ssaList);
```

Important: To persist changes made to the database, your application must call the commit method prior to deallocating the PSB, otherwise the changes are rolled back up to the last point the commit method was called.

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Deleting data in a IMS Universal DL/I driver application

Use the delete method in the PCB interface to delete existing segments in the database.

In the IMS Universal DL/I driver, the delete methods provide functionality similar to the DL/I DLET call. The delete call must be preceded by a HOLD operation. Deleting a segment causes all its child segments to be deleted. The delete method will return an IMS status code indicating the results of the DL/I operation.

The following are the general steps to delete existing segments in the database:

1. Obtain an unqualified SSAList instance from the PCB instance representing the database.
2. Optionally, you can add qualification statements to the SSAList instance. See "Specifying segment search arguments using the SSAList interface" for more information.
3. Get a Path instance by using the SSAList instance from steps 1 and 2 and calling the getPathForRetrieveReplace method.
4. Perform a Hold operation before issuing the replace call. The Hold operation can be a getUnique, getNext, or getNextWithinParent method call.
5. You can delete all the segments on the Path retrieved by step 3 or delete a subset of the segments.
   - To delete all the segments on the Path, call the PCB.delete method with no arguments.
   - If the Path retrieved by step 3 returned multiple segments from the database and you do not want to delete all the segments on the Path, use the PCB.delete method that takes an SSAList argument and pass in an unqualified SSAList for the segment where you want the deletion to begin. An exception is thrown if a qualified SSAList is provided as an argument.

IMS Universal DL/I driver delete examples

The following code fragment illustrates how delete all segments in a Path. Calling the delete method with no arguments removes all PATIENT segments and its dependent segments (ILLNESS, TREATMNT, DOCTOR, BILLING) where the patient name is "ANDREA SMITH", the ward name is "SURG", the hospital name is "ALEXANDRIA", and the patient number is "PatientNo7".

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","ILLNESS");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",SSAList.EQUALS,"ALEXANDRIA");
ssaList.addInitialQualification("WARD","WARDNAME",SSAList.EQUALS,"SURG");
ssaList.addInitialQualification("PATIENT","PATNAME",SSAList.EQUALS,"ANDREA SMITH");
ssaList.addCommandCode("PATIENT", SSAList.CC_D);
Path path = ssaList.getPathForRetrieveReplace();
if (pcb.getUnique(path, ssaList, true)) {
   if (path.getString("PATIENT", "PATNUM").equals("PatientNo7")) {
      pcb.delete();
   }
}
while (pcb.getNext(path, ssaList, true)) {
   if (path.getString("PATIENT", "PATNUM").equals("PatientNo7")) {
      pcb.delete();
   }
}
```

The following code fragment illustrates how to use delete with an unqualified SSAList. Calling the delete method with an unqualified SSAList removes all ILLNESS segments and its dependent segments (TREATMNT, DOCTOR) where the patient name is "ANDREA SMITH", the ward name is "SURGICAL", the hospital name is "ALEXANDRIA", and the patient number is "PatientNo7".

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","ILLNESS");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",SSAList.EQUALS,"ALEXANDRIA");
ssaList.addInitialQualification("WARD","WARDNAME",SSAList.EQUALS,"SURGICAL");
ssaList.addInitialQualification("PATIENT","PATNAME",SSAList.EQUALS,"ANDREA SMITH");
ssaList.markAllFieldsForRetrieval("PATIENT", true);
Path path = ssaList.getPathForRetrieveReplace();
SSAList illnessSSAList = pcb.getSSAList("ILLNESS");
if (pcb.getUnique(path, ssaList, true)) {
   if (path.getString("PATIENT", "PATNUM").equals("PatientNo7")) {
      pcb.delete(illnessSSAList);
```

```
   }
}
while (pcb.getNext(path, ssaList, true)) {
   if (path.getString("PATIENT", "PATNUM").equals("PatientNo7")) {
      pcb.delete(illnessSSAList);
   }
}
```

Important: To persist changes made to the database, your application must call the commit method prior to deallocating the PSB, otherwise the changes are rolled back up to the last point the commit method was called.

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Making batch data deletions in a IMS Universal DL/I driver application

Use the batchDelete method in the PCB interface to delete multiple existing segments in the database with one call.

The following are the general steps to delete multiple existing segments in the database with a single call:

1. Obtain an unqualified SSAList instance from the PCB instance representing the database.
2. Optionally, you can add qualification statements to the SSAList. See "Specifying segment search arguments using the SSAList" for more information.
3. Call the batchDelete method to delete the segments specified by the SSAList in the previous steps.

The following code fragment illustrates how to use the batchDelete method to remove a patient's records. The SSAList instance is set to restrict the deletion operation to remove only records where the patient name is "ANDREA SMITH", the ward name is "SURG", and the hospital name is "ALEXANDRIA".

```
SSAList ssaList = pcb.getSSAList("HOSPITAL","PATIENT");
ssaList.addInitialQualification("HOSPITAL","HOSPNAME",SSAList.EQUALS,"ALEXANDRIA");
ssaList.addInitialQualification("WARD","WARDNAME",SSAList.EQUALS,"SURG");
ssaList.addInitialQualification("PATIENT","PATNAME",SSAList.EQUALS,"ANDREA SMITH");
pcb.batchDelete(ssaList);
```

Important: To persist changes made to the database, your application must call the PSB.commit method prior to deallocating the PSB, otherwise the changes are rolled back up to the last point commit was called.

Related concepts:

"Specifying segment search arguments using the SSAList interface" on page 619

Inspecting the PCB status code and related information using the com.ibm.ims.dli.AIB interface

To inspect the PCB status code, return code, reason code, error code extension, and related information after a data access call by the IMS Universal drivers, use the com.ibm.ims.dli.AIB interface provided by the IMS Universal DL/I driver.

Typically, the IMS Universal drivers throws an exception if a call is not successful. The non-blank status codes that do not generate an exception are GD, GE, GB, GA, GK, QC, QD, and CF. In error cases, the generated exception contains the most pertinent information such as PCB status code, return code, reason code, and error

code extension. To inspect the PCB status code and related information for all the non-error cases, use the com.ibm.ims.dli.AIB interface.

The AIB instance contains all the data attributes of an IMS application interface block. The AIB instance also contains a reference to a com.ibm.ims.dli.DBPCB instance, which contains all the data attributes of a PCB instance.

When your IMS Universal drivers application makes a data access call to IMS, the application is internally making a DL/I call using a com.ibm.ims.dli.PCB instance. After each DL/I call that your application issues, IMS places a two-character status code in the DBPCB instance stored in the AIB instance for that PCB instance.

The com.ibm.ims.dli.IMSStatusCodes class contains constants for the IMS status codes. Use this helper class for comparison checking of the status code from the DL/I call.

The following code example shows how to access the AIB instance from a IMS Universal DL/I driver application:

```
try {
   psb = PSBFactory.createPSB(connSpec);
} catch (DLIException e) {
   AIB aib = e.getAib();
   if (aib != null) {
      String sc = aib.getDBPCB().getStatusCodeChars();
      String retcode = aib.getReturnCodeHex();
      String reascode = aib.getReasonCodeHex();
      System.out.println("Status code: " + sc + " Return Code: "
         + retcode + " Reason Code: " + reascode);
      }
}
```

The following code example shows how to access the AIB instance from a JDBC application. Note that you need to import the com.ibm.ims.dli package to use the AIB and DLIException objects in your Java code.

```
try {
   resultSet.updateString(1, "Harry Houdini");
} catch (SQLException e) {
   Throwable t = e.getCause();
   if (t != null && t instanceof DLIException) {
      com.ibm.ims.dli.DLIException de = (com.ibm.ims.dli.DLIException) t;
      com.ibm.ims.dli.AIB aib = de.getAib();
      if (aib != null) {
         String sc = aib.getDBPCB().getStatusCodeChars();
         String retcode = aib.getReturnCodeHex();
         String reascode = aib.getReasonCodeHex();
         System.out.println("Status code: " + sc + " Return Code: "
            + retcode + " Reason Code: " + reascode);
      }
    }
}
```

Related reference:

DL/I status code explanations (Messages and Codes)

Committing or rolling back DL/I transactions

The IMS Universal DL/I driver provides support for local transactions with the commit and rollback methods.

A local transaction consists of a unit of work with several units of recovery. A IMS Universal DL/I driver application can commit or roll back changes to the database

within a unit of recovery. In the IMS Universal DL/I driver, the local transaction is scoped to the PSB instance. No explicit call is needed to begin a local transaction. A unit of work starts when the application allocates a PSB object and obtains a connection to the database by calling the PSB.allocate method.

After the unit of work starts, the application makes DL/I calls to access the database and create, replace, insert, or delete data. The application commits the current unit of recovery by using the PSB.commit method. The commit operation instructs the database to commit all changes to the database that are made from the point when the unit of work started, or from the point after the last commit or rollback method call, whichever was most recent.

Important: To persist changes made to the database, your application must call the commit method prior to deallocating the PSB, otherwise the changes are rolled back up to the last point the commit method was called.

The application can also end the unit of recovery by calling a roll back operation using the PSB.rollback method. Calling a roll back operation causes the database to undo all changes to the database made from the start of the unit of work, or from the point after the most recent commit or rollback call.

If the PSB.commit method or the PSB.rollback method is called and the PSB instance is not deallocated, a new unit of recovery is started. The overall unit of work ends when the PSB instance is deallocated. If the PSB.commit method or the PSB.rollback method are called while they are not currently in a unit of work (either before the PSB instance is allocated or after it is deallocated), an exception is thrown.

Local transaction with a single PSB

The following example code shows a local transaction for a single PSB.

```
IMSConnectionSpec connSpec = IMSConnectionSpecFactory.createIMSConnectionSpec();
connSpec.setDatastoreName("IMS1");
connSpec.setDatastoreServer("ecdev123.svl.ibm.com");
connSpec.setPortNumber(5555);
connSpec.setMetadataURL("class://BMP266.BMP266DatabaseView");
connSpec.setUser("usr");
connSpec.setPassword("password");
connSpec.setDriverType(IMSConnectionSpec.DRIVER_TYPE_4);

PSB psb = PSBFactory.createPSB(connSpec);
psb.allocate(); // new unit of work begins
PCB pcb = psb.getPCB("PCb01");

SSAList ssa = pcb.getSSAList("HOSPITAL");
Path path = ssa.getPathForInsert("HOSPITAL");
path.setString("HOSPCODE", "R1210020000A");
path.setString("HOSPNAME", "SANTA TERESA");
pcb.insert(path);
psb.commit();  // or use psb.rollback() to undo the insert.
              // The unit of receovery ends.
```

In this example, the application makes a connection to the database and allocates a PSB. The application obtains a PCB and specifies the path to insert a new HOSPITAL record. The application then performs a DL/I operation to insert the new record into the database. At this point, the application commits the insert operation and the new record is written to the database. Alternatively, the application can roll back the insert operation to return the database to the previous state before the insert call was made. This ends the current unit of recovery.

Local transaction with multiple PSBs

When two or more PSB objects are allocated by an application, separate local transactions for each PSB may run concurrently. The following example code shows multiple local transactions with two PSBs.

```
IMSConnectionSpec connSpec = IMSConnectionSpecFactory.createIMSConnectionSpec();
connSpec.setDatastoreName("IMS1");
connSpec.setDatastoreServer("ecdev123.svl.ibm.com");
connSpec.setPortNumber(5555);
connSpec.setMetadataURL("class://BMP266.BMP266DatabaseView");
connSpec.setUser("usr");
connSpec.setPassword("password");
connSpec.setDriverType(IMSConnectionSpec.DRIVER_TYPE_4);

// create a connection to MyDB
PSB psb = PSBFactory.createPSB(connSpec);
psb.allocate(); // new unit of work begins for psb

// create another connection to MyDB.
// Note: This does not need be be a connection to the same database.
PSB psb2 = PSBFactory.createPSB(connSpec);
psb2.allocate();

pcb = psb.getPCB("PCb01");
SSAList ssa = pcb.getSSAList("HOSPITAL");
Path path = ssa.getPathForInsert("HOSPITAL");
path.setString("HOSPCODE", "R1210020000A");
path.setString("HOSPNAME", "SANTA TERESA");
pcb.insert(path);
psb.commit(); // or use psb.rollback() to undo the insert.
              // The unit of recovery for psb ends

pcb2 = psb2.getPCB("PCb01");
SSAList ssa2 = pcb2.getSSAList("HOSPITAL");
Path path2 = ssa2.getPathForInsert("HOSPITAL");
path2.setString("HOSPCODE", "R1210010000A");
path2.setString("HOSPNAME", "ALEXANDRIA");
pcb2.insert(path2);
psb2.rollback(); //or use psb2.commit() to commit the insert.
                 // The unit of recovery for psb2 ends

psb2.deallocate(); // unit of work ends for psb2
psb.deallocate(); // unit of work ends for psb
```

In this example, the application makes two connections to the same database. A PSB is allocated for the first connection and the application performs a DL/I operation to insert a new HOSPITAL record with hospital name “SANTA TERESA” into the database. Another PSB is allocated to the second connection and an insert operation is made for a new HOSPITAL record with hospital name “ALEXANDRIA”. The application then commits the changes for the first PSB and writes the new record with hospital name “SANTA TERESA” to the database. The application issues a roll back statement for the second PSB, undoing the previous insert operation for the record with hospital name “ALEXANDRIA”. Only one new record is inserted to the database: the HOSPITAL record with hospital name “SANTA TERESA”.

Configuring the IMS Universal drivers for SSL support

With type-4 connectivity, the IMS Universal drivers provide support for the Secure Sockets Layer (SSL) through the Java Secure Socket Extension (JSSE).

This information applies to type-4 connectivity only. You can use SSL support in your Java applications in either a container-managed environment with the IMS Universal Database resource adapter, or in a stand-alone environment with the IMS Universal JDBC driver and the IMS Universal DL/I driver.

Configuring the IMS Universal Database resource adapter for SSL support in a container-managed environment

To enable SSL in a container-managed environment for the IMS Universal Database resource adapter, you need to configure the SSL certificate and key management settings from your WebSphere Application Server administrative console.

Prerequisites:

- You must first set up the IBM z/OS Communications Server Application Transparent Transport Layer Security (AT-TLS) to enable SSL support on the z/OS system for IMS Connect.
- You also need to retrieve the client certificate (.crt) to your local file system where WebSphere Application Server is installed. To retrieve the certificate, from TSO, browse the `OMVSADM.CERTAUTH.CERT` member. Copy its contents into a text file on your local file system, and remove any trailing spaces. Name the file `hostname.crt`.

To configure the IMS Universal Database resource adapter for SSL support:

1. Open the WebSphere Application Server administrative console.
2. From the left pane, expand **Security -> SSL certificate and key management**.
3. Click **Key stores and certificates**.
4. Click **NodeDefaultTrustStore**.
5. Click **Signer certificates**.
6. Click **Add**.
7. In the **`Alias`** field, type a name that helps you remember that this certificate is associated with (extracted from) the server key ring file that was created when you set up AT-TLS to enable SSL on IMS Connect.
8. In the **`File name`** field, type the fully qualified path to the .crt file located on your local file system.
9. Click **OK** and then click **Save**. The trusted certificate is picked up automatically and used during the SSL handshaking process at run time.

Related tasks:

Setting up AT-TLS SSL for IMS Connect (System Definition)

Configuring IMS Universal drivers for SSL support in a stand-alone environment

To enable SSL in a stand-alone environment for the IMS Universal drivers, you need to generate and configure an SSL keystore.

Prerequisites:

- You must first set up the IBM z/OS Communications Server Application Transparent Transport Layer Security (AT-TLS) to enable SSL support on the z/OS system for IMS Connect.
- You also need to retrieve the client certificate (.crt) to your local file system. To retrieve the certificate, from TSO, browse the `OMVSADM.CERTAUTH.CERT` member.

Copy its contents into a text file on your local file system, and remove any trailing spaces. Name the file `hostname.crt`.

To configure the IMS Universal DL/I driver or the IMS Universal JDBC driver for SSL support:

1. Generate a new SSL keystore by using the Java Keytool provided by the Java SDK. This keystore file will be used as a truststore by the JRE during SSL handshaking when it creates an SSL connection to IMS. Save the keystore (.ks) file on your local file system and record its location. Set the password for the keystore and record it.

   Keystore files can contain public/private key pairs that are generated on the local system as well as public keys (in the form of certificates) that are received from remote communicating peers. When the keystore is accessed to retrieve a certificate of a communicating peer for use during SSL handshaking, the keystore file is referred to as a *truststore*.
2. Verify that the certificate has not been tampered with before importing the certificate (.crt) file into the keystore as a trusted self-signed certificate. You can do this with the Keytool by viewing the fingerprint of the local certificate and comparing it to the original that was extracted from the key ring file on the host.
3. Set the fully qualified path to the keystore file as the value for the system property javax.net.ssl.trustStore and set the keystore password as the value for the system property javax.net.ssl.trustStorePassword. Optionally, to troubleshoot any SSL-related problems, you can turn on the SSL client-side trace by setting the system property `javax.net.debug=all`. To specify the system properties from the command line, enter:

   ```
   java -Djavax.net.debug=all -Djavax.net.ssl.trustStore=myTruststore
   -Djavax.net.ssl.trustStorePassword=myTruststorePassword MyApp
   ```

Related tasks:

Setting up AT-TLS SSL for IMS Connect (System Definition)

Tracing IMS Universal drivers applications

To obtain data for diagnosing problems with the IMS Universal drivers, you can collect trace data.

Use one of the following procedures to enable tracing.

Turning on automatic tracing in JRE logging.properties file

The recommended method is to enable the trace by setting the trace level for the IMS Universal drivers loggers in the logging.properties file of your Java Runtime Environment (JRE). Using this method, the application does not need to be recompiled. The file is located on the install path of your JRE, under `\jre\lib\logging.properties`. The recommended trace level is FINEST.

To set the trace for all IMS Universal drivers loggers, add the following line to the `logging.properties` file:

```
com.ibm.ims.* = FINEST
```

To send the trace output to a file, add the following lines to your `logging.properties` file:

```
java.util.logging.FileHandler.level = FINEST
java.util.logging.FileHandler.pattern = c:/UniversalDriverTrace.txt
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

Configuring J2EE tracing

Tracing can be turned on from your J2EE application server. In WebSphere Application Server, this is configured through the administrative console. The IMS Universal Database resource adapter must be deployed on WebSphere Application Server before tracing can be configured.

To get the most detailed trace from the IMS Universal Database resource adapter, follow these steps:

1. Start the WebSphere Application Server administration console.
2. Select **Troubleshooting**.
3. Select **Logs and Trace**.
4. Select your application server from the table.
5. Under **General Properties**, select **Diagnostic Trace**.
6. Under **Additional Properties**, select **Change Log Detail Levels**.
7. Select the **Runtime** tab.
   - Make sure that the **Save runtime changes to configuration as well** check box is turned **ON**.
   - Under the **Change Log Details Levels** section, select the component **com.ibm.ims.***. This brings up the **Message and Trace levels** menu.
   - Select the message level **FINEST**.
   - Click **Apply**.
8. To save these changes for the next time the application server is started, click the **Save** link at the top of the page. WebSphere Application Server does not need to be restarted.

Programmatically enabling tracing

You can also programmatically turn on tracing in your IMS Universal drivers application. This requires the application to be recompiled.

1. Import the java.util.logging package in your application and create a logger by calling the Logger.getLogger method with the String argument "com.ibm.ims.*".
2. In your application, you can set the level of tracing for the logger by using the Logger.setLevel method. The recommended trace level is Level.FINEST.

The following sample code shows how programmatic trace is enabled for any IMS Universal drivers application.

```
private static final Logger universalLogger
   = Logger.getLogger("com.ibm.ims.*");
universalLogger.setLevel(Level.FINEST);
FileHandler fh
   = new FileHandler("C:/UniversalTrace.txt");
fh.setFormatter(new SimpleFormatter());
fh.setLevel(Level.FINEST);
universalLogger.addHandler(fh);
```

Chapter 38. Programming Java dependent regions

Use these topics to design, write, and maintain application programs for running in the Java dependent regions.

Related concepts:

Chapter 35, "IMS solutions for Java development overview," on page 539

Overview of the IMS Java dependent regions

The IMS Java dependent regions are two types of IMS dependent regions that provide a Java Virtual Machine (JVM) environment for Java applications: Java message processing (JMP) regions and Java batch processing (JBP) regions.

Important: You can host Java applications on the mainframe to access IMS from the following z/OS environments:

- JMP and JBP regions
- WebSphere Application Server for z/OS
- DB2 for z/OS stored procedures
- CICS

Use the JMP or JBP regions to host your Java application if your application is required to run in an IMS dependent region.

JMP and JBP regions can run applications written in Java, object-oriented COBOL, object-oriented PL/I, or a combination of these languages.

To access IMS message queues from your JMP and JBP applications, use the IMS Java dependent region resource adapter. To access IMS databases from your JMP and JBP applications, you can also use these IMS Universal drivers: the IMS Universal JDBC driver and the IMS Universal DL/I driver.

In addition to IMS databases, you can access DB2 for z/OS databases from your JMP and JBP applications by using the JDBC driver for DB2 for z/OS (JCC driver version 3.57.91).

Java message processing (JMP) regions

JMP regions are like message processing program (MPP) regions, but JMP regions allow the scheduling only of Java programs. In the PSB source associated with the Java program, the option `LANG=JAVA` must be specified. A JMP application is started when there is a message in the queue for the JMP application and IMS schedules the message to be processed. JMP applications, like MPP applications, are executed through transaction codes submitted by users at terminals and from other applications. Each transaction code represents a transaction that the JMP application processes.

A single application can also be started from multiple transaction codes. JMP applications, like MPP applications, are flexible in how they process transactions and where they send the output. JMP applications send any output messages back to the message queues and process the next message with the same transaction

© Copyright IBM Corp. 1974, 2015

code. The program continues to run until there are no more messages with the same transaction code. JMP applications share the following characteristics:

- They are small.
- They can produce output that is needed immediately.
- They can access IMS or DB2 data in a DB/DC environment and DB2 data in a DCCTL environment.

Java batch processing (JBP) regions

JBP regions run flexible programs that perform batch-type processing online and can access the IMS message queues for output, like non-message-driven batch message processing (BMP) applications. JBP applications are started by submitting a job with JCL or from TSO. JBP applications are like BMP applications, except that they cannot read input messages from the IMS message queue. For example, there is no `IN=` parameter in the startup procedure. Like BMP applications, JBP applications can use symbolic checkpoint and restart calls to restart the application after an abend. JBP applications can access IMS or DB2 for z/OS data in a DB/DC or DBCTL environment and DB2 for z/OS data in a DCCTL environment

Related tasks:

"IBM Enterprise COBOL for z/OS interoperability with JMP and JBP applications" on page 665

"Accessing DB2 for z/OS databases from JMP or JBP applications" on page 667

Related reference:

Chapter 37, "Programming with the IMS Universal drivers," on page 549

Programming with the IMS Java dependent region resource adapter

IMS provides a set of Java APIs called the IMS Java dependent region resource adapter to develop Java applications to run on the IMS Java dependent regions.

The IMS Java dependent region resource adapter provides Java application programs running in JMP or JBP regions with similar DL/I functionality to that provided in message processing program (MPP) and non-message driven BMP regions, such as:

- Accessing IMS message queues to read and write messages
- Performing program switches
- Commit and rollback processing
- Accessing GSAM databases
- Database recovery (CHKP/XRST)

Use the IMS Java dependent region resource adapter together with the type-2 IMS Universal JDBC driver or type-2 IMS Universal DL/I driver to perform database operations, including GSAM database access.

The following figure shows a Java application that is running in a JMP or JBP region. Database access and message processing requests are passed to the IMS Java dependent region resource adapter and type-2 IMS Universal drivers, which converts the calls to DL/I calls.

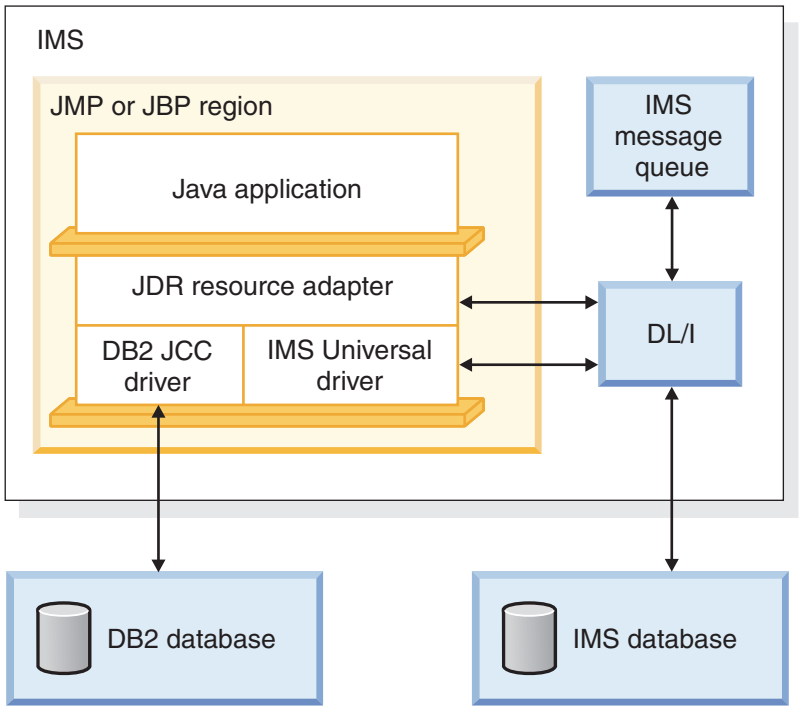


Figure 105. JMP or JBP application that is using the IMS Java dependent region resource adapter

Preparing to write a Java application with the IMS Java dependent region resource adapter

The IMS Java dependent region resource adapter is available as an SMP/E-installable driver (imsutm.jar).

Java application programs that use the IMS Java dependent region resource adapter require the Java Development Kit (JDK) 6.0 or later. They also require a way to generate the IMS database metadata, such as using the IMS Enterprise Suite Explorer for Development. The default segment encoding of the database metadata class produced by the IMS Explorer for Development is `cp1047`. To change the segment encoding, use the com.ibm.ims.base.DLIBaseSegment.setDefaultEncoding method.

Related concepts:

Chapter 6, “Gathering requirements for message processing options,” on page 99

Developing JMP applications with the IMS Java dependent region resource adapter

Java message processing (JMP) applications access the IMS message queue to receive messages to process and to send output messages.

Defining the input and output message classes

Before your JMP application can access the message queue, you must define input and output message classes by subclassing the com.ibm.ims.application.IMSFieldMessage class.

Recommendation: Use the IMS Enterprise Suite Explorer for Development to generate the necessary metadata class files from COBOL copybooks or PL/I resources when this support is available.

The com.ibm.ims.application.IMSFieldMessage and com.ibm.ims.base.DLITypeInfo class can be found in the classic Java APIs for IMS.

The IMS Java dependent region resource adapter provides the capability to process IMSFieldMessage objects.

Subclass IMSFieldMessage: input message sample code

This example code subclasses the com.ibm.ims.application.IMSFieldMessage class to make the fields in the message available to the program and creates an array of com.ibm.ims.base.DLITypeInfo objects for the fields in the message. For the DLITypeInfo class, the code identifies first the field name, then the data type, the position, and finally the length of the individual fields within the array. This allows the application to use the access functions within the IMSFieldMessage class hierarchy to automatically convert the data from its format in the message to a Java type that the application can process. In addition to the message-specific fields that it defines, the IMSFieldMessage class provides access functions that allow it to determine the transaction code and the length of the message.

This class defines an input message that accepts a 2-byte type code of a car model to query a car dealership database for available car models.

```
package dealership.application;
import com.ibm.ims.db.*;
import com.ibm.ims.base.*;
import com.ibm.ims.application.*;

/* Subclasses IMSFieldMessage to define application's input messages */
public class InputMessage extends IMSFieldMessage {

    /* Creates array of DLITypeInfo objects for the fields in message */
    final static DLITypeInfo[]fieldInfo={
        new DLITypeInfo("ModelTypeCode", DLITypeInfo.CHAR, 1, 2)
    };

    public InputMessage() {
        super(fieldInfo, 2, false);
    }
}
```

Subclass IMSFieldMessage: output message sample code

The following code example shows how to subclass the com.ibm.ims.application.IMSFieldMessage class to define an output message that displays the available car models from a type code query.

This sample code creates an array of com.ibm.ims.base.DLITypeInfo objects and then passes that array, the byte array length, and the Boolean value false, which indicates a non-SPA message, to the IMSFieldMessage constructor. For each DLITypeInfo object, you must first identify the field data type, then the field name, the field offset in the byte array, and finally the length of the byte array.

```
package dealership.application;
import com.ibm.ims.db.*;
import com.ibm.ims.base.*;
import com.ibm.ims.application.*;

/*Subclasses IMSFieldMessage to define application's output messages */
public class ModelOutput extends IMSFieldMessage {

s        /* Creates array of DLITypeInfo objects for the fields in message */
    final static DLITypeInfo[] fieldInfo={
```

```
            new DLITypeInfo("Type",         DLITypeInfo.CHAR,    1,  2),
            new DLITypeInfo("Make",         DLITypeInfo.CHAR,    3, 10),
            new DLITypeInfo("Model",        DLITypeInfo.CHAR,   13, 10),
            new DLITypeInfo("Year",         DLITypeInfo.DOUBLE, 23,  4),
            new DLITypeInfo("CityMiles",    DLITypeInfo.CHAR,   27,  4),
            new DLITypeInfo("HighwayMiles", DLITypeInfo.CHAR,   31,  4),
            new DLITypeInfo("Horsepower",   DLITypeInfo.CHAR,   35,  4)
        };

        public ModelOutput() {
            super(fieldInfo, 38,false);
        }

    }
```

JMP programming models

JMP applications can retrieve input messages from the IMS message queue, access IMS and DB2 for z/OS databases, commit or roll back transactions, and send output messages.

Creating the `main` method for a JMP application

The `main` method (`public static void main(String[] args)`) is the program entry point for all JMP and JBP applications.

A JMP application starts when IMS receives a message with a transaction code for the JMP application and schedules the message. A JMP application typically ends when there are no more messages with that transaction code to process.

JMP application `main` method code sample

The following code sample shows how to implement a JMP application to access the hospital database and send messages:

```
package hospital.ims;

import java.sql.*;
import com.ibm.ims.dli.tm.*;
import com.ibm.ims.dli.DLIException;

public static void main(String args[]) {
  try {
    Application app = null;
    MessageQueue messageQueue = null;
    IOMessage inputMessage  = null;
    IOMessage outputMessage  = null;
    Transaction tran = null;

    app = ApplicationFactory.createApplication();
    inputMessage = app.getIOMessage("class://hospital.ims.InMessage");
    outputMessage = app.getIOMessage("class://hospital.ims.OutMessage");
    messageQueue = app.getMessageQueue();
    tran = app.getTransaction();

    IMSDataSource dataSource = new IMSDataSource();
    dataSource.setMetadataURL("class://hospital.ims.HospitalDBView");
    dataSource.setDriverType(IMSDataSource.DRIVER_TYPE_2);
    dataSource.setDatastoreName("IMS1");

    Connection conn = dataSource.getConnection();
    conn.SetAutoCommit(false);
    Statement st = conn.createStatement();

    String in = new String("");
```

```
    // Returns true if a message is read from the queue
    while (messageQueue.getUnique(inputMessage)) {

      in = inputMessage.getString("Message").trim();
      if (!in.equals("")) {

    // Query the database for all hospital names
        ResultSet rs
    = st.executeQuery("SELECT HOSPNAME FROM PCB01.HOSpital");

        while (rs.next()) {

    // Return hospital name in output message
          outputMessage.setString("Message", rs.getString("HOSPNAME"));
          messageQueue.insert(outputMessage,
          MessageQueue.DEFAULT_DESTINATION);

    // Commit this transaction
          tran.commit();
  }
       conn.close();
      }
    }
  } catch (Exception e) {
     e.printStackTrace();
  }
}
```

Accessing DB2 for z/OS data from a JMP application

When a JMP application accesses only IMS data, it must open a database connection only once to process multiple transactions. However, a JMP application that accesses DB2 for z/OS data must open and close a database connection for each message that is processed.

Processing an input message in a JMP application:

A transaction begins when the application receives an input message and ends when the application commits the results from processing the message. To get an input message, the application calls the MessageQueue.getUnique method.

Processing an input message sample code

The following code example shows how an input message is processed in a JMP application.

```
import com.ibm.ims.dli.tm.*;

public static void main(String args[]) {

    conn = dataSource.getConnection(...); //Establish DB connection

    while(messageQueue.getUnique(...)){ //Get input message, which
                                          //starts transaction

        results=statement.executeQuery(...); //Perform DB processing
        ...
        messageQueue.insert(...);     //Send output messages
        ...
    }
```

```
    conn.close();                           //Close DB connection

    return;
}
```

Rolling back IMS changes in a JMP application:

A JMP application can roll back IMS changes any number of times during a transaction. A rollback call backs out all output messages to the most recent commit.

Use the com.ibm.ims.dli.tm.Transaction class to issue commit and rollback operations from your JMP application.

The following code example shows how a JMP application rolls back IMS changes.

```
import com.ibm.ims.dli.tm.*;
import java.sql.*;

public static void main(String args[]) {

   conn = dataSource.getConnection(...); //Establish DB connection
   Application app = ApplicationFactory.createApplication();
   Transaction tran = app.getTransaction();
   MessageQueue mq = app.getMessageQueue();

   while(mq.getUnique(...)){ //Get input message, which
                  //starts transaction

      results=statement.executeQuery(...); //Perform DB processing
      ...
      mq.insertMessage(...); //Send output messages
      ...
      tran.rollback(); //Roll back output messages

      results=statement.executeQuery(...); //Perform more DB processing
                 //(optional)
      ...
      mq.insert(...); //Send more output messages
                //(optional)
   }

   conn.close(); //Close DB connection
}
```

Additional message handling considerations for JMP applications

The following considerations apply to JMP applications that access the IMS message queue when handling conversational transactions, multi-segment messages, messages with repeating structures, and multiple input messages.

Conversational transactions:

The IMS Java dependent region resource adapter supports access to IMS conversational transactions.

Conversational transactions

A conversational transaction does not process the entire transaction at the same time. A conversational program divides processing into a connected series of terminal-to-program-to-terminal interactions. Use conversational processing when

one transaction contains several parts. In contrast, a nonconversational program receives a message from a terminal, processes the request, and sends a message back to the terminal.

A conversational program receives a message from a terminal and replies to the terminal, but it saves the data from the transaction in a scratchpad area (SPA). When the user at the terminal enters more data, the program has the data it saved from the last message in the SPA, so it can continue processing the request without the user at the terminal having to enter the data again.

Conversational transaction sample

The following code example shows how to write a JMP application to process a conversational transaction.

```
package mytest.jdbo;

import com.ibm.ims.dli.DLIException;
import com.ibm.ims.dli.tm.*;

public class MyConversationalSample {

  public static void main(String[] args) {
    Transaction tran = null;
    try {
      Application app
         = ApplicationFactory.createApplication();
      IOMessage spaMessage
         = app.getIOMessage("class://mytest.jdbo.SPAMessage");
      IOMessage inputMessage
         = app.getIOMessage("class://mytest.jdbo.InMessage");
      IOMessage outputMessage
         = app.getIOMessage("class://mytest.jdbo.OutMessage");
      MessageQueue msgQueue = app.getMessageQueue();
      tran = app.getTransaction();

      // Read the SPA message
      while (msgQueue.getUnique(spaMessage)) {
        // before reading the application messages.
        if (msgQueue.getNext(inputMessage)) {
          String inField
            = inputMessage.getString("Message").trim();
          try {
            int sum = (new Integer(inField)).intValue();
            spaMessage.setString("Message", "" + sum);
            msgQueue.insert(spaMessage,
              MessageQueue.DEFAULT_DESTINATION);
            outputMessage.setString("Message",
              "The initial value is: " + sum);
            msgQueue.insert(outputMessage,
              MessageQueue.DEFAULT_DESTINATION);
          } catch (NumberFormatException e) {
            if (inField.equalsIgnoreCase("stop")) {
              // End the conversation
              spaMessage.setString("Message",
                "Exit requested, so I am exiting");
spaMessage.setTransactionName(IOMessage.END_CONVERSATION_BLANKS);
msgQueue.insert(spaMessage, MessageQueue.DEFAULT_DESTINATION);
            }
          }
        }
      }
      tran.commit();
    } catch (DLIException e) {
      e.printStackTrace();
```

```
      try {
        // Roll back the transaction
        if (tran != null) {
          tran.rollback();
        }
      } catch (DLIException e1) {
        e1.printStackTrace();
      }
    }
   }
}
```

Conversational transaction sequence of events

When the message is a conversational transaction, the following sequence of events occurs:

1. IMS removes the transaction code and places it at the beginning of a message segment. The message segment is equal in length to the SPA that was defined for this transaction during system definition. The transaction code is the first segment of the input message that is made available to the program. The second through the *n*th segments from the terminal, minus the transaction code, become the remainder of the message that is presented to the application program.
2. After the conversational program prepares its reply, it inserts the SPA to IMS. The program then inserts the actual text of the reply as segments of an output message.
3. IMS saves the SPA and routes the message to the input LTERM (logical terminal).
4. If the SPA insert specifies that another program is to continue the same conversation, the total reply (including the SPA) is retained on the message queue as input to the next program. This program then receives the message in a similar form.
5. A conversational program must be scheduled for each input exchange. The other processing continues while the operator at the input terminal examines the reply and prepares new input messages.
6. To terminate a conversation, the program places blanks in the transaction code field of the SPA and inserts the SPA to IMS. To terminate a conversation when using the IMS Java dependent region resource adapter, set the SPA transaction code to the constant IOMessage.END_CONVERSATION_BLANKS.
7. The conversation can also be terminated if the transaction code in the SPA is replaced by any transaction code from a nonconversational program, and the SPA is inserted to IMS. After the next terminal input, IMS routes that message to the queue of the other program in the normal way.

Related concepts:

Conversational transactions (Communications and Connections)

Handling multi-segment messages:

Message-driven applications can have multi-segment input messages. That is, more than one message needs to be read from the message queue in order to retrieve the entire message.

The following code shows how the IOMessage and the MessageQueue classes are used to retrieve multi-segment messages:

```
//Create a message queue
MessageQueue messageQueue = app.getMessageQueue();

//Create the first input message
IOMessage input1
  = app.getIOMessage("class://InputMessage1");

//Create the second input message
IOMessage input2
  = app.getIOMessage("class://InputMessage2");

try {
  //Read the first message from the queue
  messageQueue.getUnique(input1);
  ...
 //Read additional messages from the queue
 while(messageQueue.getNext(input2)) {
  ...
} catch (DLIException e) {
  ...
}
```

Coding and accessing messages with repeating structures:

Messages with repeating structures can be defined by using the `DLITypeInfoList` class. With the `DLITypeInfoList` class, you can specify a repeating list of fields and the maximum number of times the list can be repeated. These repeating structures can contain repeating structures.

The following code example is a sample output message that contains a set of Make, Model, and Color fields:

Sample output message with repeating structures

```
public class ModelOutput extends IMSFieldMessage {
static DLITypeInfo[] modelTypeInfo = {
     new DLITypeInfo("Make", DLITypeInfo.CHAR, 1, 20),
     new DLITypeInfo("Model", DLITypeInfo.CHAR, 21, 20),
     new DLITypeInfo("Color", DLITypeInfo.CHAR, 41, 20),
     };
static DLITypeInfo[] modelTypeInfoList = {
     new DLITypeInfoList("Models", modelTypeInfo, 1, 60, 100),
     };
public ModelOutput() {
   super(modelTypeInfoList, 6004, false);
} }
```

To access the nested structures that are defined in a `DLITypeInfoList` object, use a dotted notation to specify the fields of the field within a repeating structure. For example, the “Color” field in the fourth “Models” definition in the `output` object is accessed as “Models.4.Color” within the `output` message. The following code sets the fourth “Color” in the `output` message to “Red.”

```
IOMessage output = app.getIOMessage("class://ModelOutput");
output.setString("Models.4.Color", "Red");
```

Flexible reading of multiple input messages:

JMP applications can process multiple input messages that require different input data types.

The following car dealership sample application supports requests to list models, show model details, find cars, cancel orders, and record sales. Each of these requests requires different input data.

The following steps explain how to define the messages to support these requests, and how to access the messages from the application.

1. Define the primary input message.

   The primary input message is the message that you pass to the MessageQueue.getUnique method to retrieve all of your input messages. Your primary input message must have an I/O area that is large enough to contain any of the input requests that your application might receive. It must also contain at least one field in common with all of your input messages. This common field allows you to determine the input request. In the following code example, the common field is CommandCode, and the maximum length of each message is 64 (the number passed to the IMSFieldMessage constructor):

```
public class InputMessage extends IMSFieldMessage {

    final static DLITypeInfo[] fieldInfo =
    {
       new DLITypeInfo("CommandCode",
          DLITypeInfo.CHAR, 1, 20),
    };

    public InputMessage(DLITypeInfo[] fieldInfo)
    {
        super(fieldInfo, 64, false);
    }
}
```

2. Define separate input messages for each request.

   Each of these input messages contains the same CommandCode field as its first field. Each of these input messages also uses an IMSFieldMessage constructor that takes an IMSFieldMessage object and a DLITypeInfo array. The IMSFieldMessage constructor allows you to remap the contents of the primary input message using the same type of information with each request; therefore, you do not copy the I/O area of the message, only a reference to this area. The following code example illustrates how to create the input messages for the requests `ShowModelDetails`, `FindACar`, and `CancelOrder`.

```
public class ShowModelDetailsInput extends IMSFieldMessage {
   final static DLITypeInfo[] fieldInfo = {
      new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
      new DLITypeInfo("ModelTypeCode", DLITypeInfo.CHAR, 21, 2)
   };

   public ShowModelDetailsInput(InputMessage inputMessage) {
      super(inputMessage, fieldInfo);
   }
}

public class FindACarInput extends IMSFieldMessage {
   final static DLITypeInfo[] fieldInfo = {
      new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
      new DLITypeInfo("Make", DLITypeInfo.CHAR, 21, 10),
      new DLITypeInfo("Model", DLITypeInfo.CHAR, 31, 10),
      new DLITypeInfo("Year", DLITypeInfo.CHAR, 41, 4),
      new DLITypeInfo("LowPrice", DLITypeInfo.PACKEDDECIMAL, 45, 5),
      new DLITypeInfo("HighPrice", DLITypeInfo.PACKEDDECIMAL, 50, 5),
      new DLITypeInfo("Color", DLITypeInfo.CHAR, 55, 10),
   };
   public FindACarInput(InputMessage inputMessage) {
      super(inputMessage, fieldInfo);
```

```
        }
    }

    public class CancelOrderInput extends IMSFieldMessage {
        final static DLITypeInfo[] fieldInfo = {
            new DLITypeInfo("CommandCode", DLITypeInfo.CHAR, 1, 20),
            new DLITypeInfo("OrderNumber", DLITypeInfo.CHAR, 21, 6),
            new DLITypeInfo("DealerNumber", DLITypeInfo.CHAR, 21, 6),
        };
        public CancelOrderInput(InputMessage inputMessage)
        {
            super(inputMessage, fieldInfo);
        }
    }
```

Note the following details about the previous code examples:

- The CommandCode field is defined in every message that reads the command code. If you do not define the field, you must adjust the offsets of the following fields to account for the existence of the CommandCode in the byte array. For example, you can delete the `DLITypeInfo` entry for CommandCode in the `CancelOrderInput` class, but the OrderNumber field must still start at offset 21.
- The length of the base class `InputMessage` must be large enough to contain any of its subclasses. In this example, the `InputMessage` class is 65 bytes because the fields of the `FindACarInput` method require it.
- Each `InputMessage` subclass must provide a constructor to create itself from an `InputMessage` object. This constructor uses a new constructor in the `IMSFieldMessage` class, called a *copy constructor*.

Given this design, an application can provide message-reading logic like in the following code example.

```
while (messageQueue.getUnique(inputMessage)) {

    string commandCode=inputMessage.getString("CommandCode").trim();

    if (commandCode.equals("ShowModelDetails")) {
        showModelDetails(new ShowModelDetailsInput(inputMessage));

    } else if(commandCode.equals("FindACar")) {
        findACar(new FindACarInput(inputMessage));

    } else {
        //process an error
    }

}
```

Developing JBP applications with the IMS Java dependent region resource adapter

JBP applications are similar to JMP applications, except that JBP applications do not receive input messages from the IMS message queue. Unlike batch message processing (BMP) applications, JBP applications must be non-message-driven applications.

Symbolic checkpoint and restart

Similarly to batch message processing (BMP) applications, JBP applications can use symbolic checkpoint and restart calls to restart the application after an abend. To

issue a symbolic checkpoint and restart when using the IMS Java dependent region resource adapter, use these methods of the com.ibm.ims.dli.tm.Transaction interface:

- Transaction.checkpoint()
- Transaction.restart()

These methods perform functions that are analogous to the DL/I system service calls: (symbolic) CHKP and XRST.

A JBP application connects to a database, makes a restart call, performs database processing, periodically checkpoints, and disconnects from the database at the end of the program. The program must issue a final commit before ending. On an initial application start, the Transaction.restart() method notifies IMS that symbolic checkpoint and restart is to be enabled for the application. The application then issues periodic Transaction.checkpoint() calls to take checkpoints. The Transaction.checkpoint() method allows the application to provide a com.ibm.ims.dli.tm.SaveArea object that contains one or more other application Java objects whose state is to be saved with the checkpoint.

If a restart is required, it defaults to the last checkpoint ID. The Transaction.restart() method returns a SaveArea object that contains the application objects in the same order in which they were inserted at checkpoint time. If the SaveArea object returned is null, this means there were no objects stored in the SaveArea object at checkpoint time.

Symbolic checkpoint and restart calls may also be used with GSAM data, or z/OS data sets. To restart using a basic z/OS checkpoint, you must identify the restart checkpoint.

Code sample of JBP symbolic checkpoint and restart

The following symbolic checkpoint/restart sample JBP application demonstrates the use of the checkpoint and restart functionality support with the IMS Java dependent region resource adapter.

The two symbolic checkpoint methods checkpoint() andcheckpoint(SaveArea saveArea) require the application to be restarted (in the case of any abnormal end of the program) using the 4-character constant “LAST”.

```
package samples.dealership.chkp_xrst;

import java.sql.*;
import java.io.*;

import com.ibm.ims.dli.DLIException;
import com.ibm.ims.dli.tm.Application;
import com.ibm.ims.dli.tm.ApplicationFactory;
import com.ibm.ims.dli.tm.SaveArea;
import com.ibm.ims.dli.tm.Transaction;
import com.ibm.ims.jdbc.IMSDataSource;

public class CheckpointRestartAutoSample {

    private SaveArea saveAreaOut;
    private Connection connection;
    private Transaction transaction;

    // The entry point of the application
    public static void main(String[] args) throws Exception {
        CheckpointRestartAutoSample crSample
```

```
                                  = new CheckpointRestartAutoSample();

                              crSample.setup();

                              crSample.runSample();

                              crSample.closeDown();

                          }

                          // Set up for the application:
                          //  1. Enable trace
                          //  2. Creates connection
                          void setup() throws Exception {
                              this.createConnection();
                              Application app = ApplicationFactory.createApplication();
                              this.transaction = app.getTransaction();
                          }

                          void closeDown() throws Exception {
                              // close the connection
                              connection.close();

                              // Commit the IMS DB work
                              this.transaction.commit();
                          }

                          // Creates a connection to the auto dealership database
                          void createConnection() throws Exception {
                              try {
                                  IMSDataSource ds = new IMSDataSource();
                               ds.setDriverType(IMSDataSource.DRIVER_TYPE_2);
                               ds.setMetadataURL("class://samples.dealership.AUTPSB11DatabaseView");

                                  connection = ds.getConnection();

                              } catch (SQLException e) {
                                  String errorMessage
                                     = new String("During connection creation: "
                                          + e.toString());
                                  throw new Exception(errorMessage);
                              }
                          }

                          void runSample() throws Exception {
                              // the restart call is always the first call in a
                              // checkpoint/restart application
                              saveAreaOut = this.transaction.restart();

                              // if the SaveArea object returned is null it
                              // is a normal program start, otherwise it is a restart
                              if (saveAreaOut != null) {
                                  // Check the SaveArea object to determine
                                     // where to restart from
                                  if (saveAreaOut.isEmpty()) {
                                      sqlMethod(true);
                                  } else {
                                      String str =
                                         (String)saveAreaOut.getObject(1);
                                      System.out.println("Retrieved string = "+str);
                                  }
                              } else {
                                  sqlMethod(false);
                              }
                          }

                          void sqlMethod(boolean isRestart)
```

```
                throws DLIException, SQLException {

                 String sql
                   = new String("SELECT * FROM Dealer.DealerSegment");
                 Statement statement = connection.createStatement();
                 ResultSet results = statement.executeQuery(sql);

                 // this part of the code will be executed only during a normal
                 // program start
                 if (!isRestart && results.next()) {
                     System.out.println("At first GetSegment call to the DealerDB: ");
                     System.out.println("Dealer Number = "
                        + results.getString("DealerNo"));
                     System.out.println("Dealer Name   = "
                        + results.getString("DealerName"));
                     System.out.println("DealerCity    = "
                        + results.getString("DealerCity"));
                     System.out.println("DealerZip     = "
                        + results.getString("DealerZip"));
                     System.out.println("DealerPhone   = "
                        + results.getString("DealerPhone"));
                 }

                 //String ckptid = null;
                 for (int i=1; results.next(); i++) {
                     System.out.println("GetSegment call to the DealerDB:");
                     System.out.println("Dealer Number = "
                        + results.getString("DealerNo"));
                     System.out.println("Dealer Name   = "
                        + results.getString("DealerName"));
                     System.out.println("DealerCity    = "
                        + results.getString("DealerCity"));
                     System.out.println("DealerZip     = "
                        + results.getString("DealerZip"));
                     System.out.println("DealerPhone   = "
                        + results.getString("DealerPhone"));

                // The checkpoint call, apart from storing program information,
                // causes the program to lose its position in the database
                     this.transaction.checkpoint();
                 }
             }

}
```

Rolling back changes in a JBP application

Similar to JMP applications, a JBP application can roll back database processing and output messages any number of times during a transaction. A rollback call backs out all database processing and output messages to the most recent commit.

Use the com.ibm.ims.dli.tm.Transaction class to issue commit and rollback operations from your JMP application.

Related concepts:

"Restarting your program" on page 68

Related reference:

Chapter 16, "Recovering databases and maintaining database integrity," on page 283

Accessing GSAM data from a JBP application

GSAM data are frequently referred to as z/OS data sets or as flat files. This kind of data is non-hierarchical in structure. You can access data from GSAM databases from a JBP application.

The JMP application connects to a GSAM database, performs database processing, periodically commits, and disconnects from the database at the end of the application. To access the GSAM data, you will need to supply your JBP application with the Java database metadata class for that database.

Sample metadata class for a car dealership database

The following Java code sample provides an example of the Java database metadata class.

```
package samples.dealership.gsam;

import com.ibm.ims.db.*;
import com.ibm.ims.base.*;

public class AUTOGSAMDatabaseView extends DLIDatabaseView {

// This class describes the data view of PSB: AUTOGSAM
// PSB AUTOGSAM has database PCBs with 8-char PCBNAME or label:
// AUTOLPCB
// PCBGSAMG
// PCBGSAML

// The following describes Segment:
  // DEALER ("DEALER") in PCB: AUTOLPCB ("AUTOLPCB")
static DLITypeInfo[] AUTOLPCBDEALERArray= {
new DLITypeInfo("DLRNO", DLITypeInfo.CHAR, 1, 4,
  "DLRNO", DLITypeInfo.UNIQUE_KEY),
new DLITypeInfo("DLRNAME", DLITypeInfo.CHAR,
  5, 30, "DLRNAME"),
new DLITypeInfo("CITY", DLITypeInfo.CHAR,
  35, 10, "CITY"),
new DLITypeInfo("ZIP", DLITypeInfo.CHAR,
  45, 10, "ZIP"),
new DLITypeInfo("PHONE", DLITypeInfo.CHAR,
  55, 7, "PHONE")
};
static DLISegment AUTOLPCBDEALERSegment= new DLISegment
("DEALER","DEALER",AUTOLPCBDEALERArray,61);

// The following describes Segment: MODEL ("MODEL")
  // in PCB: AUTOLPCB ("AUTOLPCB")
static DLITypeInfo[] AUTOLPCBMODELArray= {
new DLITypeInfo("MODKEY", DLITypeInfo.CHAR, 3, 24,
  "MODKEY", DLITypeInfo.UNIQUE_KEY),
new DLITypeInfo("MODTYPE", DLITypeInfo.CHAR, 1, 2, "MODTYPE"),
new DLITypeInfo("MAKE", DLITypeInfo.CHAR, 3, 10, "MAKE"),
new DLITypeInfo("MODEL", DLITypeInfo.CHAR, 13, 10, "MODEL"),
new DLITypeInfo("YEAR", DLITypeInfo.CHAR, 23, 4, "YEAR"),
new DLITypeInfo("MSRP", DLITypeInfo.CHAR, 27, 5, "MSRP"),
new DLITypeInfo("COUNT1", DLITypeInfo.CHAR, 32, 2, "COUNT")
};
static DLISegment AUTOLPCBMODELSegment= new DLISegment
("MODEL","MODEL",AUTOLPCBMODELArray,37);

// An array of DLISegmentInfo objects follows
  // to describe the view for PCB: AUTOLPCB ("AUTOLPCB")
static DLISegmentInfo[] AUTOLPCBarray = {
new DLISegmentInfo(AUTOLPCBDEALERSegment,DLIDatabaseView.ROOT),
new DLISegmentInfo(AUTOLPCBMODELSegment,0),
```

```
};

// Warning: PCB: PCBGSAMG has no SENSEGS
// The following describes GSAM Record:
  // JAVGSAM1 ("JAVGSAM1") in PCB: PCBGSAMG ("GSAMRead")
static DLITypeInfo[] PCBGSAMGJAVGSAM1Array= {
new DLITypeInfo("DealerNo", DLITypeInfo.INTEGER, 1, 4),
new DLITypeInfo("DealerName", DLITypeInfo.CHAR, 5, 30),
new DLITypeInfo("ModelType", DLITypeInfo.CHAR, 35, 2),
new DLITypeInfo("ModelKey", DLITypeInfo.CHAR, 37, 24),
new DLITypeInfo("Make", DLITypeInfo.CHAR, 37, 10),
new DLITypeInfo("Model", DLITypeInfo.CHAR, 47, 10),
new DLITypeInfo("Year", "yyyy", DLITypeInfo.DATE, 57, 4),
new DLITypeInfo("MSRP", "S999999V99", DLITypeInfo.PACKEDDECIMAL, 61, 5),
new DLITypeInfo("Counter", DLITypeInfo.SMALLINT, 66, 2)
};
static GSAMRecord PCBGSAMGRecord= new GSAMRecord
("PCBGSAMGRecord",PCBGSAMGJAVGSAM1Array,80);

// An array of DLISegmentInfo objects follows
  // to describe the view for PCB: PCBGSAMG ("GSAMRead")
static DLISegmentInfo[] PCBGSAMGarray = {
new DLISegmentInfo(PCBGSAMGRecord, DLIDatabaseView.ROOT)
};

// Warning: PCB: PCBGSAML has no SENSEGS
// The following describes GSAM Record:
  // JAVGSAM1 ("JAVGSAM1") in PCB: PCBGSAML ("GSAMLoad")
static DLITypeInfo[] PCBGSAMLJAVGSAM1Array= {
new DLITypeInfo("DealerNo", DLITypeInfo.INTEGER, 1, 4),
new DLITypeInfo("DealerName", DLITypeInfo.CHAR, 5, 30),
new DLITypeInfo("ModelType", DLITypeInfo.CHAR, 35, 2),
new DLITypeInfo("ModelKey", DLITypeInfo.CHAR, 37, 24),
new DLITypeInfo("Make", DLITypeInfo.CHAR, 37, 10),
new DLITypeInfo("Model", DLITypeInfo.CHAR, 47, 10),
new DLITypeInfo("Year", "yyyy", DLITypeInfo.DATE, 57, 4),
new DLITypeInfo("MSRP", "S999999V99",
  DLITypeInfo.PACKEDDECIMAL, 61, 5),
new DLITypeInfo("Counter", DLITypeInfo.SMALLINT, 66, 2)
};
static GSAMRecord PCBGSAMLRecord= new GSAMRecord
("PCBGSAMLRecord",PCBGSAMLJAVGSAM1Array,80);

// An array of DLISegmentInfo objects follows
  // to describe the view for PCB: PCBGSAML ("GSAMLoad")
static DLISegmentInfo[] PCBGSAMLarray = {
new DLISegmentInfo(PCBGSAMLRecord, DLIDatabaseView.ROOT)
};



// Constructor
public AUTOGSAMDatabaseView() {
super("2.0","AUTOGSAM", "AUTOLPCB", "AUTOLPCB",
  AUTOLPCBarray);
addDatabase("GSAMRead", "PCBGSAMG", PCBGSAMGarray);
addDatabase("GSAMLoad", "PCBGSAML", PCBGSAMLarray);
} // end AUTOGSAMDatabaseView constructor

} // end AUTOGSAMDatabaseView class definition
```

Sample JBP application for accessing a GSAM database

The following code example is a JBP application that relies on the previous code sample to access GSAM data.

```
package samples.dealership.gsam;

import java.io.*;
import java.util.Properties;
import java.math.BigDecimal;

import com.ibm.ims.dli.*;
import com.ibm.ims.dli.tm.*;

/**
* This is an auto dealership sample application
* demonstrating the use of the
* GSAM database functionality support in
* the IMS Java dependent region resource adapter.
*
*/
public class GSAMAuto {
private final String readOnlyGSAMPCB
  = new String("GSAMRead");
private final String writeOnlyGSAMPCB
  = new String("GSAMLoad");

private PSB psb;

/**
* The entry point of the application
*/
public static void main(String[] args) {
GSAMAuto gsamLoadSample = new GSAMAuto();
if (System.getProperty("com.ibm.ims.jdbcenvironment")
  == null) {
Properties properties = System.getProperties();
properties.put("com.ibm.ims.jdbcenvironment", "IMS");
}

try {
gsamLoadSample.setup();
} catch (Exception e) {
e.printStackTrace();
}

try {
gsamLoadSample.runSample();

gsamLoadSample.closeDown();
} catch (Throwable e) {
e.printStackTrace();
}

}

/**
* This method does the set up for the application:
* 1. Enable trace
* 2. Creates dbConnection
* 3. Creates GSAMConnection object
* @throws IOException
* @throws SecurityException
* @throws DLIException
*/
void setup() throws SecurityException,
  IOException, DLIException {
IMSConnectionSpec cSpec
  = IMSConnectionSpecFactory.createIMSConnectionSpec();
cSpec.setDatastoreName("IMS1");
cSpec.setDriverType(IMSConnectionSpec.DRIVER_TYPE_2);
cSpec.setMetadataURL("class://samples.dealership.gsam.AUTOGSAMDatabaseView");
```

```
psb = PSBFactory.createPSB(cSpec);
}

/**
* This method does the clean up before application exit.
* 1. Commits the database work done. IMS Java dependent
* regions require all applications to commit before exiting.
* @throws DLIException
*
* @exception Exception
*/
void closeDown() throws DLIException {
try {
Application app = ApplicationFactory.createApplication();
Transaction transaction = app.getTransaction();

// Always commit any work before exiting
transaction.commit();
} catch (DLIException e) {
System.out.println("IMS commit failed. Reason: "
  + e.toString());
throw e;
}
}

/**
* Demonstrates how to write to and read from a
* GSAM database. Also shows different data types
* being stored into the GSAM database using the
* internal data conversion methods.
*/
void runSample() {

final int dealerNo = 1171;
final String dealerName = "ABC Autos";
final String modelType = "LX";
final String make = "Santro";
final String model = "Zen";
final java.sql.Date year
  = java.sql.Date.valueOf("2011-05-18");
final BigDecimal msrp = new BigDecimal(17750.00);
final short count = (short) 8;

try {
GSAMPCB pcb1 = psb.getGSAMPCB(this.writeOnlyGSAMPCB);

Path myGSAMRecord = pcb1.getPathForInsert();

// Set values to individual fields in a GSAM record
myGSAMRecord.setInt("DealerNo", dealerNo);
myGSAMRecord.setString("DealerName", dealerName);
myGSAMRecord.setString("ModelType", modelType);
myGSAMRecord.setString("Make", make);
myGSAMRecord.setString("Model", model);
myGSAMRecord.setDate("Year", year);
myGSAMRecord.setBigDecimal("MSRP", msrp);
myGSAMRecord.setShort("Counter", count);

// Insert the GSAM record data
  // and save the RSA of the record
RSA rsa = pcb1.insert(myGSAMRecord);

// Close the GSAM database explicitly
  // for writing/loading data
pcb1.close();

// Open a GSAM Connection to write the GSAM dataset
```

```
GSAMPCB pcb2 = psb.getGSAMPCB(this.readOnlyGSAMPCB);

// Read the GSAM record data using
  // the RSA stored earlier
Path gsamRecord = pcb2.getUnique(rsa);

// Print the GSAM data
if (gsamRecord != null) {
System.out.println("Dealer Number: "
  + gsamRecord.getInt("DealerNo"));
System.out.println("Dealer Name: "
  + gsamRecord.getString("DealerName"));
System.out.println("Model Type: "
  + gsamRecord.getString("ModelType"));
System.out.println("Make: "
  + gsamRecord.getString("Make"));
System.out.println("Model: "
  + gsamRecord.getString("Model"));
System.out.println("Year: "
  + gsamRecord.getDate("Year"));
System.out.println("MSRP: "
  + gsamRecord.getBigDecimal("MSRP"));
System.out.println("Counter: "
  + gsamRecord.getShort("Counter"));

System.out.println
 ("\nSuccessful completion of GSAM sample application");
} else {
System.out.println("GSAM DB is empty");
}

} catch (DLIException e) {
System.out.println
 ("GSAM sample failed. Reason: " + e.toString());
}
}
}
```

Related concepts:

Chapter 19, “Processing GSAM databases,” on page 305

Related reference:

“GSAM coding considerations” on page 312

Issuing synchronous callout requests from a Java dependent region

IMS provides support for synchronous callout functionality from Java message processing (JMP) or Java batch processing (JBP) applications through an IMS implementation of the Java Message Service (JMS).

To use the JMP and JBP support for synchronous callout, the IMS Enterprise Suite JMS API jms.jar file must be on your classpath. To download the IMS Enterprise Suite JMS API, go to the following web URL: http://www-01.ibm.com/software/data/ims/enterprise-suite/index.html

The IMS implementation of JMS is limited to supporting the Point-to-Point (PTP) messaging domain only. In addition, support is only provided for non-transacted QueueSession objects with Session.AUTO_ACKNOWLEDGE mode.

If the JMP or JBP application attempts to call any JMS method not supported by IMS or pass any unsupported argument to JMS method calls, a JMSException exception is thrown.

To send a message using the JMP and JBP support for synchronous callout and synchronously receive a response:

1. Create a com.ibm.ims.jms.IMSQueueConnectionFactory object.
2. Create a JMS QueueConnection instance by calling the createQueueConnection method on the IMSQueueConnectionFactory object.
3. Create a JMS QueueSession instance by calling the createQueueSession method on the QueueConnection instance. In the method call, you must set the input parameter values to false and Session.AUTO_ACKNOWLEDGE to specify that the generated QueueSession instance is non-transacted and runs in AUTO_ACKNOWLEDGE mode.
4. Create a queue identity by calling the createQueue method on the QueueSession instance. In the method call, you must set the input parameter value to the OTMA descriptor name for the synchronous callout operation.
5. Create a JMS QueueRequestor instance and pass in the QueueSession instance from step 3 and the Queue instance from step 4 as input parameters to the QueueRequestor constructor method.
6. Create a TextMessage instance by calling the createTextMessage method on the QueueSession instance from step 3. Set the string containing the message data.
7. To send the message and retrieve a response, call the request method on the QueueRequestor object from step 5. In the method call, pass in the TextMessage instance from step 6. You need to cast the return value from the request method call to a TextMessage instance. If the call is successful, the return value is the response to the synchronous callout request.

The following code shows how to write a simple JMP or JBP application that sends a message to an external application and synchronously receive a response message. In the example, an IMSQueueConnectionFactory instance is created with a timeout value of 10 seconds and with 128 KB of space allocated to hold response messages.

```
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueRequestor;
import javax.jms.QueueSession;
import javax.jms.Session ;
import javax.jms.TextMessage;
import com.ibm.ims.jms.IMSQueueConnectionFactory;

public class IMS_Sample
{
    public static void main(String argv[])
    {
        IMSQueueConnectionFactory jmsConnectionFactory
           = new IMSQueueConnectionFactory();
        QueueConnection jmsConnection = null;
        QueueSession jmsQueueSession = null;
        Queue jmsQueue = null;
        QueueRequestor jmsQueueRequestor = null;

        try {
            jmsConnectionFactory.setTimeout(1000);
                        // set the timeout to 10 seconds
            jmsConnectionFactory.setResponseAreaLength(128000);
                        // allocate 128k to hold the response message
            jmsConnection = jmsConnectionFactory.createQueueConnection();
            jmsQueueSession
            = jmsConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
      // set session to be non-transacted and in AUTO_ACKNOWLEDGE mode
            jmsQueue = jmsQueueSession.createQueue("OTMDEST1");
```

```
                // pass in the OTMA descriptor name

                    jmsQueueRequestor
                       = new QueueRequestor(jmsQueueSession, jmsQueue);
                    TextMessage sendMsg = jmsQueueSession.createTextMessage();
                    sendMsg.setText("MyMessage");
                    System.out.println("Sending message: "+sendMsg.getText());
                    TextMessage replyMsg
                       = (TextMessage)jmsQueueRequestor.request(sendMsg);

                    System.out.println("\nReceived message: "+replyMsg.getText());
                } catch (JMSException e) {
                    e.printStackTrace();
                }
            }
        }
```

Related tasks:

"Implementing the synchronous callout function" on page 475

Related reference:

Java Message Service API Tutorial

Java Platform Enterprise Edition, v5.0 API Specifications

Program switching in JMP and JBP applications

IMS allows you to switch programs in JMP and JBP applications. You can perform immediate program switches in JMP and JBP applications, and you can also make a deferred program switch in a conversational JMP application.

Related concepts:

"Sending messages to other IMS application programs" on page 411

Immediate program switching for JMP and JBP applications

The IMS Java dependent region resource adapter supports immediate program switching in JMP and JBP applications. An immediate program switch passes the conversation directly to another conversational program that is specified by an alternate PCB.

When an application makes an immediate program switch, the first `MessageQueue.insert` call sends the SPA to the other conversational program, but subsequent `MessageQueue.insert` calls will send messages to the new program. The program does not return or respond to the original terminal.

The setAlternatePCBName method of the com.ibm.ims.dli.tm.MessageDestinationSpec class sets the name of the alternate PCB for the program switch. The setAlternatePCBName method issues the DL/I `CHNG` call.

To make an immediate program switch in a JMP or JBP application:

1. Call the MessageDestinationSpec.setAlternatePCBName method to set the name of the alternate PCB.
2. Call the `MessageQueue.insert` method to send the message to the alternate PCB.

Code sample of immediate program switching

The following code sample demonstrates how immediate program switching is performed in a JMP application.

```
package sample.jmp;

import com.ibm.ims.dli.tm.Application;
import com.ibm.ims.dli.tm.ApplicationFactory;
import com.ibm.ims.dli.tm.IOMessage;
import com.ibm.ims.dli.tm.MessageDestinationSpec;
import com.ibm.ims.dli.tm.MessageQueue;
import com.ibm.ims.dli.tm.Transaction;

public class SampleJMPImmediatePgmSwitch {
  private static IOMessage outputMessage = null;
  private static MessageQueue msgQueue = null;
  private static Application app = null;
  private static IOMessage inputMessage  = null;

  public static void main(String[] args) {
     try {
        app = (Application) ApplicationFactory.createApplication();
 msgQueue = (MessageQueue) app.getMessageQueue();
 inputMessage
          = app.getIOMessage("class://sample.jmp.InMessage");
 outputMessage
          = app.getIOMessage("class://sample.jmp.OutMessage");

       //Define Message Destinations Specs
 MessageDestinationSpec mds2
          = new MessageDestinationSpec();
 mds2.setAlternatePCBName("TPPCB1");
 mds2.setDestination("JAVTRANJ");

       String in = new String("");
       while (msgQueue.getUnique(inputMessage)){
         in = inputMessage.getString("Message").trim();
   if(in.equalsIgnoreCase("ImmediatePGMSwitch1")){
            outputMessage.setString("Message",
               "Running ImmediatePGMSwitch1 Call");
     msgQueue.insert(outputMessage,
               MessageQueue.DEFAULT_DESTINATION);

       // Insert Message to JAVTRANJ TPPCB1: DLIWithCommit
            outputMessage.setString("Message",
               "Insert Message to JAVTRANJ TPPCB1");
     msgQueue.insert(outputMessage,
               MessageQueue.DEFAULT_DESTINATION);

     outputMessage.setString("Message", "DLIWithCommit");
     outputMessage.setTransactionName("JAVTRANJ");

       // Insert message to JAVTRANJ
      msgQueue.insert(outputMessage, mds2);

       // Commit transaction
            Transaction tran = app.getTransaction();
       tran.commit();
   } else {
            outputMessage.setString("Message",
               "Invalid input - valid input is 'ImmediatePGMSwitch1'");
     msgQueue.insert(outputMessage,
               MessageQueue.DEFAULT_DESTINATION);
     Transaction tran = app.getTransaction();
            tran.commit();
   }
       }
      } catch (Exception e) {
```

```
            e.printStackTrace();
         }
      }
   }
```

To perform immediate program switching in a JBP application, the steps are similar to a JMP application. In the main module, setup a MessageDestinationSpec instance then issue an insert call to another transaction. For example:

```
MessageDestinationSpec mds2 = new MessageDestinationSpec();
mds2.setAlternatePCBName("TPPCB1");
mds2.setDestination("JAVTRANJ");
...
outputMessage.setString("Message", "Some Message");
outputMessage.setTransactionName("JAVTRANJ");
msgQueue.insert(outputMessage, mds2);
```

Related concepts:

"Passing the conversation to another conversational program" on page 423

Deferred program switching for conversational JMP applications

You can make a deferred program switch in a conversational JMP application. A deferred program switch changes the transaction code in the scratchpad area (SPA) before the SPA is returned to IMS. When an application makes a deferred program switch, the application replies to the terminal and passes the conversation to another conversational application.

Use the `setTransactionName(String)` method of the `com.ibm.ims.dli.tm.IOMessage` class to specify the transaction code in the SPA.

To make a deferred program switch in a conversational JMP application:

1. Call the `insert(IOMessage)` method to send the output message to the terminal.
2. Call the `setTransactionName(String)` method to set the name of the transaction code in the SPA.
3. Call the `insert(IOMessage)` method to send the SPA to IMS.

Code sample of deferred program switching

The following code sample demonstrates how deferred program switching is performed in a JMP application.

```
package sample.jmp;

import com.ibm.ims.dli.tm.Application;
import com.ibm.ims.dli.tm.ApplicationFactory;
import com.ibm.ims.dli.tm.IOMessage;
import com.ibm.ims.dli.tm.MessageDestinationSpec;
import com.ibm.ims.dli.tm.MessageQueue;
import com.ibm.ims.dli.tm.Transaction;


public class SampleJMPDeferredPGM {
   private static IOMessage spaMessage = null;
   private static MessageQueue msgQueue = null;
   private static Application app = null;
   private static IOMessage inputMessage  = null;
   private static Transaction tran = null;

   public static void main(String[] args) {
      try {
         app = ApplicationFactory.createApplication();
  spaMessage
         = app.getIOMessage("class://sample.jmp.SPAMessage");
```

```
inputMessage
        = app.getIOMessage("class://sample.jmp.InMessage");
msgQueue = app.getMessageQueue();
tran = app.getTransaction();

MessageDestinationSpec mds
        = new MessageDestinationSpec();
mds.setAlternatePCBName("TPPCB1");
mds.setDestination("IVTCM");

       String in = new String("");
while (msgQueue.getUnique(spaMessage)) {
          if (msgQueue.getNext(inputMessage)) {
      in = inputMessage.getString("Message").trim();
      if (in.equalsIgnoreCase("DeferredPGMSwitch2")) {
         inputMessage.setString("Message", spaMessage.getString("Message"));
  msgQueue.insert(inputMessage, MessageQueue.DEFAULT_DESTINATION);

   // Setting Deferred Program Switch
  inputMessage.setString("Message", "Setting Deferred Program Switch");
  msgQueue.insert(inputMessage, MessageQueue.DEFAULT_DESTINATION);

  spaMessage.setString("Message", "SampleJMPDeferredPGM");
  spaMessage.setTransactionName("IVTCM");
  msgQueue.insert(spaMessage, mds);

  inputMessage.setString("Message", "SampleJMPDeferredPGM Completed");
  msgQueue.insert(inputMessage, MessageQueue.DEFAULT_DESTINATION);

  tran.commit();
 } else {
  inputMessage.setString("Message", spaMessage.getString("Message"));
  msgQueue.insert(inputMessage, MessageQueue.DEFAULT_DESTINATION);

  inputMessage.setString("Message",
                    "Input Message was not 'DeferredPGMSwitch2'");
  msgQueue.insert(inputMessage, MessageQueue.DEFAULT_DESTINATION);

 tran.commit();
 }
     }
  }
 } catch(Exception e){
   e.printStackTrace();

 }
   }
}
```

Related concepts:

"Passing the conversation to another conversational program" on page 423

IBM Enterprise COBOL for z/OS interoperability with JMP and JBP applications

With the IBM Enterprise COBOL for z/OS support for COBOL and Java language interoperability, you can write Java and Object-Oriented (OO) COBOL applications that execute in a Java dependent region and invoke existing COBOL programs.

With this support, you can:

- Call an object-oriented (OO) COBOL application from a Java application by building the frontend application, which processes messages, in Java, and the back end, which processes databases, in OO COBOL.

- Build an OO COBOL application containing a main routine that can invoke Java routines.

You can access COBOL code in a JMP or JBP region because Enterprise COBOL provides object-oriented language syntax that enables you to:

- Define classes with methods and data implemented in COBOL
- Create instances of Java and COBOL classes
- Invoke methods on Java and COBOL objects
- Write classes that inherit from Java classes or other COBOL classes
- Define and invoke overloaded methods

In IBM Enterprise COBOL for z/OS programs, you can call the services provided by the JNI to obtain Java-oriented capabilities in addition to the basic OO capabilities available directly in the COBOL language.

In IBM Enterprise COBOL for z/OS classes, you can code CALL statements that interface with procedural COBOL programs. Therefore, COBOL class definition syntax can be especially useful for writing wrapper classes for procedural COBOL logic, enabling existing COBOL code to be accessed from Java.

Java code can create instances of COBOL classes, invoke methods of these classes, and can extend COBOL classes.

Related Reading: For details building applications that use IBM Enterprise COBOL for z/OS and that run in an IMS dependent region, see *Enterprise COBOL for z/OS Programming Guide*.

Related concepts:

"Overview of the IMS Java dependent regions" on page 641

IBM Enterprise COBOL for z/OS backend applications in a JMP or JBP region

When you define an object-oriented (OO) COBOL class and compile it with the IBM Enterprise COBOL for z/OS compiler, the compiler generates a Java class definition with native methods and the object code to implement the native methods. After compiling the class, you can create an instance and invoke the methods of the compiled class from a Java program that runs in a JMP or JBP region.

For example, you can define an OO COBOL class with the appropriate DL/I call in COBOL to access an IMS database.

To make the implementation of this class available to a Java application running with IMS:

1. Compile the COBOL class with the IBM Enterprise COBOL for z/OS compiler to generate a Java source file, which contains the class definition, and an object module, which contains the implementation of the native methods.
2. Compile the generated Java source file with the Java compiler to create the application class file.
3. Link the object module into a dynamic link library (DLL) in the HFS file (.so).
4. Update the application class path (ibm.jvm.application.class.path) for the JMP or JBP region to allow access to the Java class file.
5. Update the library path for the JMP or JBP region to allow access to the DLL.

IBM Enterprise COBOL for z/OS frontend applications in a JMP or JBP region

The object-oriented syntax of IBM Enterprise COBOL for z/OS enables you to build COBOL applications with a `main` method, which can be run directly in a JMP or JBP region.

The JMP or JBP region locates, instantiates, and invokes the `main` method of an OO COBOL application in the same way it does for the `main` method of a Java application.

You can write an application for an JMP or JBP region entirely with OO COBOL, but a more likely use for a frontend COBOL application is to call a Java routine from a COBOL application.

When running within the JVM of an JMP or JBP region, the IBM Enterprise COBOL for z/OS runtime support automatically locates and uses the JVM to invoke methods on Java classes.

A frontend OO COBOL application with a main routine that runs in a JMP or JBP region has the same requirements as a Java program that runs in a JMP or JBP region.

Accessing DB2 for z/OS databases from JMP or JBP applications

A JMP or JBP application can access DB2 for z/OS Version 8 and DB2 for z/OS Version 9 databases by using the JDBC driver for DB2 for z/OS (JCC driver version 3.57.91).

Attention: If you access a DB2 for z/OS database using both Java and COBOL in the same application, you might experience unexpected behavior, but only if the commit or rollback processing is done in COBOL while active cursors are in the Java portion.

The JMP or JBP region that the application is running in must also be defined with DB2 for z/OS attached by the DB2 Recoverable Resource Manager Services attachment facility (RRSAF). Unlike other dependent regions, JMP and JBP regions do not use the External Subsystem Attach Facility (ESAF).

Accessing DB2 for z/OS data from a JMP or JBP application is like accessing IMS data. When writing a JMP or JBP application that accesses DB2 for z/OS data, consider both the differences from IMS database access and the differences from accessing DB2 for z/OS data in other environments:

- You must create a DB2 plan for each PSB (typically each Java application) that is used to access DB2 for z/OS.
- You can have only one active DB2 for z/OS connection open at any time.
- If you are using the type-2 JDBC drivers for DB2 for z/OS, you must use the default connection URL in the application program. For example, `jdbc:db2os390:` or `db2:default:connection`.
- If you are using the type-4 DB2 JDBC drivers, you can use a specific connection URL in the application program.
- To commit or roll back work, use the Transaction.commit method or the Transaction.rollback method.

  - For JMP applications, the Transaction.commit method commits all work, including SQL calls. Calling the Transaction.commit and Transaction.rollback methods does not automatically reset the connection to DB2 for z/OS. The connection to DB2 for z/OS is reset when you issue a MessageQueue.getUnique call.
  - For JBP applications, the Transaction.commit method commits SQL calls.
- Because RRSAF is the coordinator, you cannot use the Connection.setAutoCommit or Connection.commit method of the JDBC driver for DB2 for z/OS.

Related concepts:

"Overview of the IMS Java dependent regions" on page 641

Related tasks:

Preparing your system to use the DB2 Attach Facility (Communications and Connections)

DB2: Installing the IBM Data Server Driver for JDBC and SQLJ

Chapter 39. Programming with the classic Java APIs for IMS

Use these topics to design, write, and maintain application programs for IMS Version 11 using the classic Java APIs for IMS.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

Related concepts:

Chapter 35, "IMS solutions for Java development overview," on page 539

Programming enterprise Java applications with classic Java APIs for IMS resource adapters

IMS provides two resource adapters based on the classic Java APIs for IMS for enterprise Java applications: the IMS DB resource adapter and the IMS DB distributed resource adapter.

Recommendation: Because the IMS Universal Database resource adapter is built on industry standards and open specifications, and provides more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal Database resource adapter to develop Java EE applications that access IMS from WebSphere Application Server.

- Use the IMS DB resource adapter to create enterprise Java applications to run on WebSphere Application Server for z/OS and access IMS databases when WebSphere Application Server for z/OS and IMS are on the same logical partition (LPAR).
- Use the IMS DB distributed resource adapter to develop and deploy enterprise applications that run on non-z/OS platforms and access IMS databases remotely.

Accessing IMS data from WebSphere Application Server for z/OS with the classic Java APIs for IMS

This information covers how Enterprise JavaBeans (EJBs) hosted on WebSphere Application Server for z/OS can access IMS data using the IMS DB resource adapter based on the classic Java APIs for IMS.

Recommendation: Because the IMS Universal Database resource adapter is built on industry standards and open specifications, and provides more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal Database resource adapter to develop Java EE applications that access IMS from WebSphere Application Server.

The following figure shows how an EJB accesses IMS data using the IMS DB resource adapter. JDBC or IMS hierarchical database interface for Java calls are passed to the IMS DB resource adapter, which converts the calls to DL/I calls. The IMS DB resource adapter passes these calls to Open Database Access (ODBA), which uses the database resource adapter (DRA) to access the DL/I region in IMS.

© Copyright IBM Corp. 1974, 2015

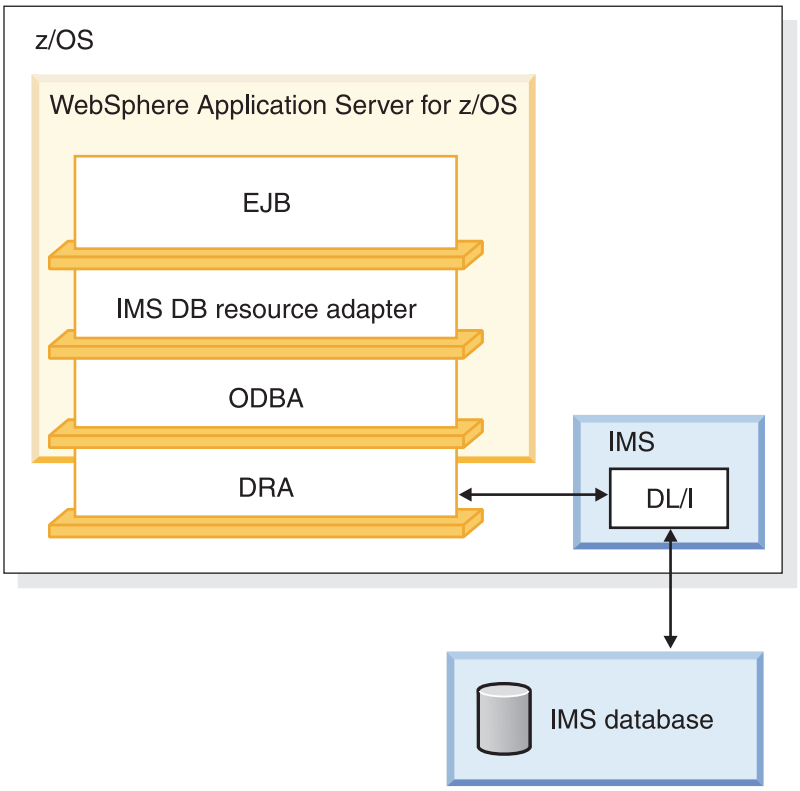


Figure 106. WebSphere Application Server for z/OS EJB using the IMS DB resource adapter

For more details on configuring ODBA and DRA to access IMS, see the topic "Loading and running the ODBA and DRA modules in the z/OS application region" in *IMS Version 11 Communications and Connections*.

Related concepts:

"Distributed and local connectivity with the IMS Universal drivers" on page 550

Bean-managed EJB programming model

In bean-managed EJBs, you programmatically define the transaction boundaries. To define an EJB as bean-managed, set the transaction-type property, which is in the ejb-jar.xml file of the EJB jar file, to Bean. You must manage the scope of the transaction by using either the javax.transaction.UserTransaction or java.sql.Connection interface.

Transaction demarcation using the javax.transaction.UserTransaction interface

The programming model applies either to Java applications that run on WebSphere Application Server for z/OS or on WebSphere Application Server for distributed platforms. With the javax.transaction.UserTransaction interface, you can define when the scope of the transaction begins and ends, and when the transaction commits or rolls back.

The EJB container supplies the EJB with a `javax.ejb.SessionContext` object that allows the `javax.transaction.UserTransaction` interface to perform the required operations to manage the transaction.

```
try {
    // Use the javax.ejb.SessionContext set by the EJB container to instantiate
    // a new UserTransaction
    javax.transaction.UserTransaction userTransaction =
```

```
                                        sessionContext.getUserTransaction();

    // Begin the scope of this transaction
    userTransaction.begin();

    // Perform JNDI lookup to obtain the data source (the IVP datasource for
    // example) and cast
    javax.sql.DataSource dataSource = (javax.sql.DataSource)
                      initialContext.lookup("java:comp/env/jdbc/IMSIVP");

    // Get a connection to the data source
    java.sql.Connection connection = dataSource.getConnection();

    // Create an SQL statement using the connection
    java.sql.Statement statement = connection.createStatement();

    // Acquire a result set by executing the query using the statement
    java.sql.ResultSet results = statement.executeQuery(...);

    // Commit and complete the scope of this transaction
    userTransaction.commit();

    // Close the connection
    connection.close();

} catch (Throwable t) {

    // If an exception occurs, roll back the transaction
    userTransaction.rollback();

    // Close the connection
    connection.close();
}
```

Transaction demarcation using the java.sql.Connection interface

This information covers the transaction processing support for the java.sql.Connection interface provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

With the java.sql.Connection interface, you commit or roll back a transaction that is started by the creation of a data source connection. An EJB bean for IMS on the server side automatically starts a transaction if one does not exist when a connection is created. You can then use this connection to commit or roll back the transaction without using the javax.transaction.UserTransaction interface.

This programming model applies only to applications that run on WebSphere Application Server on a non-z/OS platform and that use the remote database services of the classic Java APIs for IMS. Use this programming model only if you do not use the javax.transaction.UserTransaction interface.

When you perform the JNDI lookup, specify `"java:comp/env/sourceName"`, where `sourceName` is the name of the data source.

```
try {
    // Perform JNDI lookup to obtain the data source (the IVP data source
    // for example) and cast
    javax.sql.DataSource dataSource = (javax.sql.DataSource)
                      initialContext.lookup("java:comp/env/imsjavaRDSIVP");
```

```
    // Get a connection to the data source and begin the transaction scope
    java.sql.Connection connection = dataSource.getConnection();

    // Create an SQL statement using the connection
    java.sql.Statement statement = connection.createStatement();

    // Acquire a result set by executing the query using the statement
    java.sql.ResultSet results = statement.executeQuery(...);

    // Commit and complete the scope of this transaction
    connection.commit();

    // Close the connection
    connection.close();

} catch (Throwable t) {

    // If an exception occurs, roll back the transaction
    connection.rollback();

    // Close the connection
    connection.close();
}
```

Container-managed EJB programming model

In container-managed EJBs, the container manages the transaction demarcation.

The demarcation is defined in the ejb-jar.xml file of the EJB. To define an EJB as container-managed, set the **transaction-type** property, which is in the ejb-jar.xml file of the EJB jar file, to Container. Because the container manages the transaction demarcation, this programming model does not have any transaction logic.

The following code shows how to write a container-managed EJB:

```
try {

    // Perform JNDI lookup to obtain the data source (the IVP data source
    // for example) and cast
    javax.sql.DataSource dataSource = (javax.sql.DataSource)
                          initialContext.lookup("java:comp/env/jdbc/IMSIVP");

    // Get a connection to the data source
    java.sql.Connection connection = dataSource.getConnection();

    // Create an SQL statement using the connection
    java.sql.Statement statement = connection.createStatement();

    // Acquire a result set by executing the query using the statement
    java.sql.ResultSet results = statement.executeQuery(...);

    // Close the connection
    connection.close();

} catch (Throwable t) {

    // Close the connection
    connection.close();
}
```

Servlet programming model

Similarly to the bean-managed EJBs, the servlet programming model uses the UserTransaction interface to begin, commit, or roll back the transaction.

Because the servlet resides outside of the EJB container and cannot use an `EJBContext` object, the initial context requires an additional JNDI lookup to locate and instantiate the `UserTransaction` interface.

```
try {
    // Establish an initial context to manage the environment
    //properties and JNDI names
    javax.naming.InitialContext initialContext = new InitialContext();

    // Locate and instantiate a UserTransaction object that is associated with
    // the initial context using JNDI
    javax.transaction.UserTransaction userTransaction = (UserTransaction)
                                          ic.lookup("java:comp/UserTransaction");

    // Begin the scope of this transaction
    userTransaction.begin();

    // Perform JNDI lookup to obtain the data source (the IVP data source
    // for example) and cast
    javax.sql.DataSource dataSource = (javax.sql.DataSource)
                               initialContext.lookup("java:comp/env/jdbc/IMSIVP");

    // Get a connection to the datasource
    java.sql.Connection connection = dataSource.getConnection();

    // Create an SQL statement using the connection
    java.sql.Statement statement = connection.createStatement();

    // Acquire a result set by executing the query using the statement
    java.sql.ResultSet results = statement.executeQuery(...);

    // Commit and complete the scope of this transaction
    userTransaction.commit();

    // Close the connection
    connection.close();

} catch (Throwable t) {

    // If an exception occurs, roll back the transaction
    userTransaction.rollback();

    // Close the connection
    connection.close();
}
```

Requirements for WebSphere Application Server for z/OS with the classic Java APIs for IMS

This information covers programming requirements for WebSphere Application Server for z/OS when using the classic Java APIs for IMS.

Recommendation: Because the IMS Universal Database resource adapter is built on industry standards and open specifications, and provides more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal Database resource adapter to develop Java EE applications that access IMS from WebSphere Application Server.

The following programming requirements apply to Enterprise JavaBeans hosted on WebSphere Application Server for z/OS that access IMS databases using the classic Java APIs for IMS .

- The classic Java APIs for IMS do not support component-managed signon.

- The classic Java APIs for IMS do not support shared connections.
- The java.sql.Connection object must be acquired, used, and closed within a transaction boundary.
- A global transaction must exist before you create a `Connection` object from a connection with the IMS classic JDBC driver. Either specify container-demarcated transactions in the EJB deployment descriptor or explicitly begin a global transaction by calling the javax.transaction.UserTransaction interface before creating a connection with the IMS classic JDBC driver.

Deployment descriptor requirements for the classic Java APIs for IMS

This information covers the Enterprise JavaBeans (EJB) or servlet deployment descriptor requirements for the classic Java APIs for IMS.

Recommendation: Because the IMS Universal Database resource adapter is built on industry standards and open specifications, and provides more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal Database resource adapter to develop Java EE applications that access IMS from WebSphere Application Server.

The deployment descriptor for an EJB or servlet has certain requirements for the classic Java APIs for IMS. In an EJB, the deployment descriptor is the file ejb-jar.xml. In a servlet, the deployment descriptor is the file web.xml.

You must have a `resource-ref` element in the deployment descriptor. The `resource-ref` element describes external resources. In the `resource-ref` element, you must have the following elements:

```
<res-type>javax.sql.DataSource</res-type>
<res-sharing-scope>Unshareable</res-sharing-scope>
```

The `<res-type>javax.sql.DataSource</res-type>` element specifies the type of data source. The `<res-sharing-scope>Unshareable</res-sharing-scope>` element specifies that the connections are not shareable.

The following example is a resource-ref element from an EJB deployment descriptor:

```
<resource-ref>
 <res-ref-name>jdbc/DealershipSample</res-ref-name>
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Container</res-auth>
 <res-sharing-scope>Unshareable</res-sharing-scope>
</resource-ref>
```

Programming Java applications in DB2 for z/OS stored procedures with the classic Java APIs for IMS

You can write a DB2 for z/OS stored procedure to access an IMS database when DB2 for z/OS and IMS are on the same logical partition (LPAR).

Accessing IMS data from DB2 for z/OS stored procedures using the classic Java APIs for IMS

This information covers accessing IMS databases from a DB2 for z/OS stored procedure by using the classic Java APIs for IMS, ODBA, and DRA.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

The following figure shows a DB2 for z/OS stored procedure that is using the classic Java APIs for IMS, ODBA, and DRA to access IMS databases.

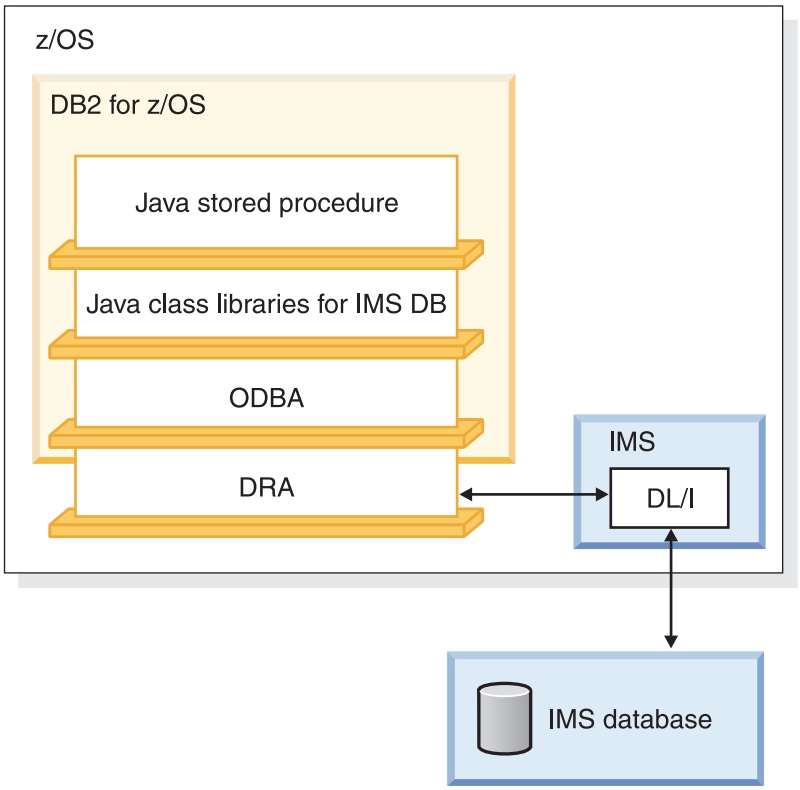


Figure 107. DB2 for z/OS stored procedure that uses the classic Java APIs for IMS

To deploy a Java stored procedure on DB2 for z/OS, you must configure the classic Java APIs for IMS, ODBA, and DRA. For details on completing these tasks, see the "Loading and running the ODBA and DRA modules in the z/OS application region" topic in *IMS Version 11 Communications and Connections*.

Related concepts:

"Distributed and local connectivity with the IMS Universal drivers" on page 550

DB2 for z/OS stored procedures programming model with the classic Java APIs for IMS

This information describes the classic Java APIs for IMS programming model for accessing IMS from DB2 for z/OS stored procedures.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

The stored procedure must perform the following tasks in the order listed. An example is given for each step:

1. Load the IMS classic JDBC driver:
   ```
   Class.forName("com.ibm.ims.db.DLIDriver");
   ```
2. Create an IMS connection with the IMS classic JDBC driver:
   ```
   connection = DriverManager.getConnection
           ("jdbc:dli:package.DatabaseViewName/DRAname");
   ```
3. Create a statement:
   ```
   Statement statement = connection.createStatement();
   ```
4. Query the IMS database:
   ```
   ResultSet results = statement.executeQuery(query);
   ```
5. Create the ResultSetConverter object and call the getDB2ResultSet method:
   ```
   ResultSetConverter converter = new ResultSetConverter();
   result2[0] = converter.getDB2ResultSet(results);
   ```

   Note: In the `ResultSetConverter` class, you must specify whether to use a declared global temporary table or a created global temporary table. The `ResultSetConverter` uses this temporary table in DB2 to convert the result set.
6. Close the connection:
   ```
   connection.close();
   ```

Programming Java applications for CICS with the classic Java APIs for IMS

You can write applications with the classic Java APIs for IMS that run on CICS Transaction Server for z/OS and access IMS databases when CICS Transaction Server for z/OS and IMS are on the same logical partition (LPAR).

Accessing IMS data from CICS with the classic Java APIs for IMS

The following information covers how a JCICS application can access IMS data using the classic Java APIs for IMS.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

The following figure shows how a JCICS application accesses an IMS database using the database resource adapter (DRA) and the classic Java APIs for IMS.

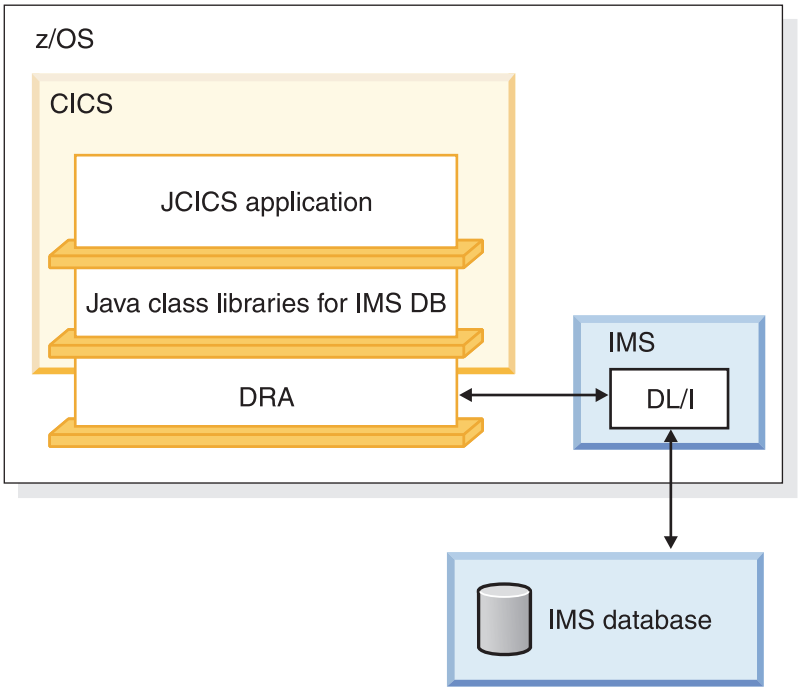


Figure 108. CICS application that uses the classic Java APIs for IMS

To deploy a Java stored procedure on CICS Transaction Server for z/OS, you must configure the classic Java APIs for IMS and DRA. For details on completing this task see the "Loading and running the ODBA and DRA modules in the z/OS application region" topic in *IMS Version 11 Communications and Connections*.

Related concepts:

"Distributed and local connectivity with the IMS Universal drivers" on page 550

CICS programming model with the classic Java APIs for IMS

The following information describes the classic Java APIs for IMS programming model for accessing IMS from CICS Transaction Server for z/OS.

Recommendation: Because the IMS Universal drivers are built on industry standards and open specifications, and provide more flexibility and improved support for connectivity, data access methods, and transaction processing options, use the IMS Universal drivers for developing your Java applications to access IMS.

The following programming model outlines the supported structure for JCICS applications that use the classic Java APIs for IMS. The model is not complete, but it shows the normal flow of the application for both the JDBC and SSA access methods.

In a CICS environment, only one PSB can be allocated at a time. Therefore, an application can have only one active connection with the IMS classic JDBC driver at a time. The application must close the connection before it opens another connection with the IMS classic JDBC driver.

```
public static void main(CommAreaHolder cah) {  //Receives control

    conn = DriverManager.getConnection(...);   //Establish DB connection

    repeat {
```

```
        results = statement.executeQuery(...); //Perform DB processing
        ...
        //send output to terminal
    }

    conn.close();                              //Close DB connection
    return;
}
```

Programming with the IMS classic JDBC driver

The IMS classic JDBC driver provides support for writing JDBC applications for CICS, DB2 store procedures, and WebSphere Application Server for z/OS environments.

The IMS classic JDBC driver supports a selected subset of the full facilities of the JDBC 2.1 API. To maintain performance, the IMS classic JDBC driver is designed to support a subset of SQL keywords that allow the IMS classic JDBC driver to perform only certain operations. Some of these SQL keywords have specific IMS usage requirements.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

This information uses the Dealership sample applications that is shipped with the classic Java APIs for IMS to describe how to use the IMS classic JDBC driver to access an IMS database.

Related concepts:

"Programming with the IMS Universal JDBC driver" on page 581

Related reference:

"Generating the runtime Java metadata class" on page 557

Sample PSB, DBD and DLIModel IMS Java report

To create the Java metadata class for a PSB, provide the application PSB source and related DBD source files to the IMS Enterprise Suite DLIModel utility plug-in so that the utility can generate the metadata class.

The following example shows the PSB for the sample dealership application.

Sample PSB for the sample dealership application

```
DLR_PCB1 PCB TYPE=DB,DBDNAME=DEALERDB,PROCOPT=GO,KEYLEN=42
         SENSEG NAME=DEALER,PARENT=0
         SENSEG NAME=MODEL,PARENT=DEALER
         SENSEG NAME=ORDER,PARENT=MODEL
         SENSEG NAME=SALES,PARENT=MODEL
         SENSEG NAME=STOCK,PARENT=MODEL
         PSBGEN PSBNAME=DLR_PSB,MAXQ=200,LANG=JAVA
         END
```

The following example shows the physical DBD that is referenced by the PSB for the sample dealership application.

DBD for the dealership sample database

```
DBD NAME=DEALERDB,ACCESS=(HDAM,OSAM),RMNAME=(DFSHDC40.1.10)
SEGM NAME=DEALER,PARENT=0,BYTES=94,
FIELD NAME=(DLRNO,SEQ,U),BYTES=4,START=1,TYPE=C
FIELD NAME=DLRNAME,BYTES=30,START=5,TYPE=C
SEGM NAME=MODEL,PARENT=DEALER,BYTES=43
FIELD NAME=(MODTYPE,SEQ,U),BYTES=2,START=1,TYPE=C
FIELD NAME=MAKE,BYTES=10,START=3,TYPE=C
FIELD NAME=MODEL,BYTES=10,START=13,TYPE=C
FIELD NAME=YEAR,BYTES=4,START=23,TYPE=C
FIELD NAME=MSRP,BYTES=5,START=27,TYPE=P
SEGM NAME=ORDER,PARENT=MODEL,BYTES=127
FIELD NAME=(ORDNBR,SEQ,U),BYTES=6,START=1,TYPE=C
FIELD NAME=LASTNME,BYTES=25,START=50,TYPE=C
FIELD NAME=FIRSTNME,BYTES=25,START=75,TYPE=C
SEGM NAME=SALES,PARENT=MODEL,BYTES=113
FIELD NAME=(SALDATE,SEQ,U),BYTES=8,START=1,TYPE=C
FIELD NAME=LASTNME,BYTES=25,START=9,TYPE=C
FIELD NAME=FIRSTNME,BYTES=25,START=34,TYPE=C
FIELD NAME=STKVIN,BYTES=20,START=94,TYPE=C
SEGM NAME=STOCK,PARENT=MODEL,BYTES=62
FIELD NAME=(STKVIN,SEQ,U),BYTES=20,START=1,TYPE=C
FIELD NAME=COLOR,BYTES=10,START=37,TYPE=C
FIELD NAME=PRICE,BYTES=5,START=47,TYPE=C
FIELD NAME=LOT,BYTES=10,START=52,TYPE=C
DBDGEN
FINISH
END
```

The DLIModel utility plug-in generates a subclass of DLIDatabaseView from the PSB and DBD. It also produces a report, called the DLIModel IMS Java report, that provides information about the metadata class.

The following example shows a DLIModel IMS Java report. The intent of the report is to assist application developers with writing IMS applications using the Java class libraries. The report provides a detailed view of the database structure and all segments and fields within that database.

Sample DLIModel IMS Java report for the dealership sample database

```
DLIModel IMS Java report
========================
Class: DealerDatabaseView in package: com.ibm.ims.tooling
generated for PSB: AUTPSB11

================================================
PCB: DealershipDB
================================================
Segment: Dealer
Field: DealerNumber Type=CHAR Length=4         ++ Primary Key Field ++
Field: DealerName Type=CHAR Length=30
Field: DealerAddress Type=CHAR Length=50
Field: YTDSales Type=PACKEDDECIMAL Type Qualifier=S9(18)
================================================
    Segment: Model
    Field: ModelTypeCode Type=CHAR Length=2    ++ Primary Key Field ++
    Field: CarMake Type=CHAR Length=10            (Search Field)
    Field: CarModel Type=CHAR Length=10         (Search Field)
    Field: CarYear Type=CHAR Length=4           (Search Field)
    Field: Price Type=CHAR Length=5             (Search Field)
    Field: EPACityMilage Type=CHAR Length=4
    Field: EPAHighwayMilage Type=CHAR Length=4
    Field: Horsepower Type=CHAR Length=4
    ================================================
```

```
Segment: Order
Field: OrderNumber Type=CHAR Length=6   ++ Primary Key Field ++
Field: PurchaserLastName Type=CHAR Length=25   (Search Field)
Field: PurchaserFirstName Type=CHAR Length=25  (Search Field)
Field: Options Type=CHAR Length=30
Field: Price Type=ZONEDDECIMAL Type Qualifier=99999
Field: OrderDate Type=CHAR Length=8
Field: SerialNo Type=CHAR Length=8
Field: DeliverDate Type=CHAR Length=8
================================================
Segment: Sales
Field: DateSold Type=CHAR Length=8     ++ Primary Key Field ++
Field: PurchaserLastName Type=CHAR Length=25    (Search Field)
Field: PurchasetFirstName Type=CHAR Length=25  (Search Field)
Field: StockVINumber Type=CHAR Length=20       (Search Field)
Field: PurchaserAddress Type=CHAR Length=25
Field: SoldBy Type=CHAR Start=84 Length=10
================================================
Segment: Stock
Field: StockVINumber Type=CHAR Length=20   ++ Primary Key Field ++
Field: Color Type=CHAR Length=10             (Search Field)
Field: Price Type=ZONEDDECIMAL Type Qualifier=99999
Field: Lot Type=CHAR Length=10               (Search Field)
Field: DateIn Type=CHAR Length=8
Field: DateOut Type=CHAR Length=8
```

The DLIModel IMS Java report provides you with the following information:

- The name of the metadata class (DealerDatabaseView in this example) to use when you establish a connection to the database.
- The hierarchy of segments for each PCB.
- The fields within each segment, which are specified by the DBD, by any COBOL copybooks, or by control statements. For example, the fields *DealerAddress* and *YTDSales* in the Dealer segment are added fields.
- The names of PCBs, segments, and fields to use in your JDBC calls. These names can be alias names that are assigned to the IMS entities. Alias names are more representative and intuitive identifiers for your Java application to use rather than the 8-character names in the PSB and DBDs. In the example, the name `DealershipDB` replaces the PCB name `DLR_PCB1` from the PSB. A comparison of the names of the segments and the fields in the report with their names in the DBD shows that they have all been assigned more meaningful names.
- The data types of fields. The data types of the fields are based on the simple TYPE property of fields in the DBD and the DLIModel utility plug-in control statements. For example, the field *YTDSales* in the Dealer segment is type PACKEDDECIMAL with a type qualifier (format descriptor) of S9(18).
- The fields in each segment, which are identified as primary or secondary index fields, search fields, or other fields.

Related concepts:

IMS Enterprise Suite DLIModel utility plug-in overview

Data transformation support for JDBC

The IMS JDBC drivers provide data transformation on behalf of client applications. When provided with the information from the Java database metadata class, the libraries are able to internally convert data from one datatype to another.

Supported JDBC data types

The following table lists the supported Java data types for each JDBC data type.

Table 103. Supported JDBC data types

JDBC data type	Java data type	Length
BIGINT	long	8 bytes
BINARY	byte[]	1 - 32 KB
BIT	Boolean	1 byte
CHAR	java.lang.String	1 - 32 KB
CLOB	java.sql.Clob	Application-defined
DATE	java.sql.Date	Application-defined
DOUBLE	double	8 bytes
FLOAT	float	4 bytes
INTEGER	int	4 bytes
PACKEDDECIMAL	java.math.BigDecimal	1 - 10 bytes
SMALLINT	short	2 bytes
TIME	java.sql.Time	Application-defined
TIMESTAMP	java.sql.Timestamp	Application-defined
TINYINT	byte	1 byte
ZONEDDECIMAL	java.math.BigDecimal	1 - 19 bytes

Methods for retrieving and converting data types

With the IMS classic JDBC driver and the IMS Universal JDBC driver, you can use the ResultSet interface (java.sql.ResultSet) to retrieve and convert the data from the type that is defined in the database metadata to the type that is required by your Java application. Similarly, with the IMS Universal DL/I driver, you can use the Path interface to perform data retrieval and conversion to Java data types.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

The following table shows the available get methods in the ResultSet interface (for the IMS classic JDBC driver and the IMS Universal JDBC driver) or the Path interface (for the IMS Universal DL/I driver) for accessing data of a certain Java data type.

The "No Truncation or Data Loss" column indicates the data types that are designed to be accessed with the given getXXX method. No truncation or data loss occurs when using those methods for those data types. The data types that are in the “Legal without Data Integrity” column are all other legal calls; however, data integrity cannot be ensured when using the given getxxx method to access those data types. If a data type is not in either column, using the given getXXX method for that data type will result in an exception.

Table 104. ResultSet.getXXX and Path.getXXX methods to retrieve data types

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getByte	TINYINT UTINYINT	SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getShort	SMALLINT USMALLINT	TINYINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getInt	INTEGER UINTEGER	TINYINT SMALLINT BIGINT FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getLong	BIGINT UBIGINT	TINYINT SMALLINT INTEGER FLOAT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getFloat	FLOAT	TINYINT SMALLINT INTEGER BIGINT DOUBLE BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]

Table 104. ResultSet.getXXX and Path.getXXX methods to retrieve data types (continued)

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getDouble	DOUBLE	TINYINT SMALLINT INTEGER BIGINT FLOAT BIT CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getBoolean	BIT	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE CHAR VARCHAR PACKEDDECIMAL[1] ZONEDDECIMAL[1]
getString	CHAR VARCHAR	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT PACKEDDECIMAL[1] ZONEDDECIMAL[1] BINARY DATE TIME TIMESTAMP
getBigDecimal	BINARY[3] PACKEDDECIMAL[1] ZONEDDECIMAL[1]	TINYINT SMALLINT INTEGER BIGINT FLOAT DOUBLE BIT CHAR VARCHAR
getClob	CLOB[2]	all others result in an exception
getBytes	BINARY	all others result in an exception
getDate	DATE	CHAR VARCHAR TYIMESTAMP
getTime	TIME	CHAR VARCHAR TIMESTAMP

Table 104. ResultSet.getXXX and Path.getXXX methods to retrieve data types (continued)

ResultSet.getXXX Method or Path.getXXX Method	Data Type (any not listed result in an exception)	
	No Truncation or Data Loss	**Legal without Data Integrity**
getTimestamp	TIMESTAMP	CHAR VARCHAR DATE TIME

Note:

1. PACKEDDECIMAL and ZONEDDECIMAL are data type extensions for the IMS classic JDBC driver , the IMS Universal JDBC driver, and the IMS Universal DL/I driver. All other types are standard SQL types defined in SQL92. **Restriction:** PACKEDDECIMAL and ZONEDDECIMAL data types do not support the Sign Leading or Sign Separate modes. For these two data types, sign information is always stored with the Sign Trailing method.
2. The CLOB data type is supported only for the retrieval and storage of XML data.
3. The BINARY data type is valid only for decimal data used with a binary type converter.

If the field type is either PACKEDDECIMAL or ZONEDDECIMAL, the type qualifier is the COBOL PICTURE string that represents the layout of the field. All COBOL PICTURE strings that contain valid combinations of 9s, Ps, Vs, and Ss are supported. Expansion of PICTURE strings is handled automatically. For example, '9(5)' is a valid PICTURE string. For zoned decimal numbers, the decimal point can also be used in the PICTURE string. `PIC 9(06)V99 COMP` and `PIC 9(06)V99 COMP-4` are valid PICTURE clauses for BINARY decimal data.

If the field contains DATE, TIME, or TIMESTAMP data, the type qualifier specifies the format of the data. For example, a type qualifier of *ddMMyyyy* indicates that the data is formatted as follows:

```
11122011 is December 11, 2011
```

For DATE and TIME types, all formatting options in the java.text.SimpleDateFormat class are supported.

For the TIMESTAMP type, the formatting option 'f' is available for nanoseconds. TIMESTAMP can contain up to nine 'f's and replaces the 'S' options for milliseconds. Instead, 'fff' indicates milliseconds of precision. An example TIMESTAMP format is as follows:

```
yyyy-mm-dd hh:mm:ss.fffffffff
```

COBOL copybook types that map to Java data types

Because data in IMS is not strongly typed, you can use COBOL copybook types to map your IMS data to Java data types.

The following table describes how COBOL copybook types are mapped to both DLITypeInfo constants in the DLIDatabaseView class and Java data types.

Table 105. Mapping from COBOL formats to DLITypeInfo constants and Java data types

Copybook format	DLITypeInfo constant	Java data type
PIC X	CHAR	`java.lang.String`

Table 105. Mapping from COBOL formats to DLITypeInfo constants and Java data types (continued)

Copybook format	DLITypeInfo constant	Java data type
PIC 9 BINARY[1]	See "DLITypeInfo constants and Java data types based on the PICTURE clause".[2]	See "DLITypeInfo constants and Java data types based on the PICTURE clause".[2]
COMP-1	FLOAT	float
COMP-2	DOUBLE	double
PIC 9 COMP-3[3]	PACKEDDECIMAL	`java.math.BigDecimal`
PIC 9 DISPLAY[4]	ZONEDDECIMAL	`java.math.BigDecimal`

Notes:

1. Synonyms for BINARY data items are COMP and COMP-4. A `PIC 9(06)V99` statement with COMP or COMP-4 is used for binary decimal data.
2. For BINARY data items, the DLITypeInfo constant and Java type depend on the number of digits in the PICTURE clause.The table "DLITypeInfo constants and Java data types based on the PICTURE clause" describes the type based on PICTURE clause length.
3. PACKED-DECIMAL is a synonym for COMP-3.
4. If the USAGE clause is not specified at either the group or elementary level, it is assumed to be DISPLAY.

The following table shows the DLITypeInfo constants and the Java data types based on the PICTURE clause.

Table 106. DLITypeInfo constants and Java data types based on the PICTURE clause

Digits in PICTURE clause	Storage occupied	`DLITypeInfo` constant	Java data type
1 through 2	1 byte	TINYINT UTINYINT	byte
1 through 4	2 bytes	SMALLINT USMALLINT	short
5 through 9	4 bytes	INTEGER UINTEGER	int
10 through 18	8 bytes	BIGINT UBIGINT	long

The following table shows examples of specific copybook formats mapped to DLITypeInfo constants.

Table 107. Copybook formats mapped to DLITypeInfo constants

Copybook format	DLITypeInfo constant
PIC X(25)	CHAR
PIC 9(02) COMP	UTINYINT
PIC S9(04) COMP	SMALLINT
PIC 9(04) COMP	USMALLINT
PIC S9(06) COMP-4	INTEGER
PIC 9(06) COMP-4	UINTEGER

Table 107. Copybook formats mapped to DLITypeInfo constants (continued)

Copybook format	DLITypeInfo constant
PIC 9(06)V99 COMP or COMP-4	BINARY
PIC S9(12) BINARY	BIGINT
PIC 9(12) BINARY	UBIGINT
COMP-1	FLOAT
COMP-2	DOUBLE
PIC S9(06)V99	ZONEDDECIMAL
PIC 9(06).99	ZONEDDECIMAL
PIC S9(06)V99 COMP-3	PACKEDDECIMAL

Connections to IMS databases

The following information covers connecting to IMS using the IMS classic JDBC driver provided in the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

When you are retrieving a JDBC `Connection` object, provide the name of the `DLIDatabaseView` subclass.

When the following code is executed, `DLIDriver`, a class in `com.ibm.ims.db`, registers itself with the JDBC `DriverManager` object:

```
Class.forName("com.ibm.ims.db.DLIDriver");
```

When the following code is executed, the JDBC `DriverManager` object determines which of the registered drivers supports the supplied string:

```
connection = DriverManager.getConnection
("jdbc:dli:dealership.application.DealerDatabaseView");
```

Because the supplied string begins with `jdbc:dli:`, the JDBC `DriverManager` object locates the `DLIDriver` instance and requests that it create a connection.

JDBC interfaces supported by the IMS classic JDBC driver

The following information covers the JDBC interfaces supported by the IMS classic JDBC driver provided in the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

Limitations will apply to the IMS implementation of the following interfaces that are implemented with JDBC 2.1:

java.sql.Connection

`java.sql.Connection` is an object that represents the connection to the database. A `Connection` reference is retrieved from the `DriverManager` object that is implemented in the `java.sql` package. The `DriverManager` object obtains a `Connection` reference by querying its list of registered `Driver`

instances until it finds one that supports the universal resource locator (URL) that is passed to the DriverManager.getConnection method.

Restriction: IMS does not support the local, connection-based commit scope that is defined in the JDBC model. Therefore, the IMS implementation of the methods Connection.commit, Connection.rollback, and Connection.setAutoCommit result in an SQL exception when these methods are called.

The following code example shows the sample dealership application code that establishes a connection to the dealership sample database:

```
connection = DriverManager.getConnection
("jdbc:dli:dealership.application.DealerDatabaseView");
```

java.sql.DatabaseMetaData

The DatabaseMetaData interface defines a set of methods for querying information about the database, including capabilities that the database might or might not support. The class is provided for tool developers and is normally not used in client programs. Much of the functionality is specific to relational databases and is not implemented for DL/I databases.

java.sql.Driver

The Driver interface itself is not usually used in client applications, although an application must dynamically load a particular Driver implementation by name. One of the first lines in an IMS JDBC program for IMS access must be:

```
Class.forName("com.ibm.ims.db.DLIDriver");
```

This code loads the JDBC driver for IMS and causes the Driver implementation to register itself with the DriverManager object so that the driver can later be found by DriverManager.getConnection. The Driver implementation creates and returns a Connection object to the DriverManager object. The JDBC driver for IMS is not fully JDBC-compliant and the Driver object method jdbcCompliant returns a value of false.

java.sql.Statement

A Statement interface is returned from the Connection.createStatement method. The Statement class and its subclass, PreparedStatement, define the interfaces that accept SQL statements and return tables as ResultSet objects. The code to create a Statement object is as follows:

```
Statement statement = connection.createStatement();
```

Restriction: The IMS implementation of the Statement interface does not support:

- Named cursors. Therefore, the method Statement.setCursorName throws an SQL exception.
- Aborting a DL/I operation. Therefore, the method Statement.cancel throws an SQL exception.
- Setting a time-out for DL/I operations. Therefore, the methods Statement.setQueryTimeout and Statement.getQueryTimeout throw SQL exceptions.

java.sql.ResultSet

The ResultSet interface defines an iteration mechanism to retrieve the data in the rows of a table, and to convert the data from the type defined in the database to the type required in the application. For example, ResultSet.getString converts an integer or decimal data type to an instance of a Java String. The following code returns a ResultSet object:

```
ResultSet results = statement.executeQuery(queryString);
```

Rather than building a complete set of results after a query is run, the IMS implementation of `ResultSet` interface retrieves a new segment occurrence each time the method `ResultSet.next` is called.

Restriction: The IMS implementation of `ResultSet` does not support:

- Returning data as an ASCII stream. Therefore the method `ResultSet.getAsciiStream` throws an SQL exception.
- Named cursors. Therefore, the method `ResultSet.getCursorName` throws an SQL exception.
- The method `ResultSet.getUnicodeStream`, which is deprecated in JDBC 2.1.

java.sql.ResultSetMetaData
The `java.sql.ResultSetMetaData` interface defines methods to provide information about the types and properties in a `ResultSet` object. It includes methods such as `getColumnCount`, `isSigned`, `getPrecision`, and `getColumnName`.

java.sql.PreparedStatement
The `PreparedStatement` interface extends the `Statement` interface, adding support for pre-compiling an SQL statement (the SQL statement is provided at construction instead of execution), and for substituting values in the SQL statement (for example, `UPDATE Suppliers SET Status = ? WHERE City = ?`).

SQL keywords and extensions for the IMS classic JDBC driver

The following information covers using SQL keywords and extensions supported by the IMS classic JDBC driver provided in the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

SQL keywords supported by the IMS JDBC drivers

The SQL support provided by the IMS classic JDBC driver and the IMS Universal JDBC driver is based on the SQL-92 standard for relational database management systems.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

If you use a SQL keyword as a name for a PCB, segment, or field, your JDBC application program will throw an error when it attempts an SQL query. These keywords are not case-sensitive.

The following SQL keywords are supported by the IMS JDBC drivers:

ALL
AND
AS
ASC
AVG
COUNT
DELETE

DESC
DISTINCT
FROM
GROUP BY
INSERT
INTO
MAX
MIN
OR
ORDER BY
SELECT
SET
SUM
UPDATE
VALUES
WHERE

Related reference:

"SELECT statement usage" on page 690

"FROM clause usage" on page 692

"WHERE clause usage" on page 693

"Portable SQL keywords restricted by the IMS Universal JDBC drivers" on page 595

Portable SQL keywords restricted by the classic IMS JDBC drivers

If you use any of the following SQL keywords as a name for a PCB, segment, or field, your JDBC application will receive an error when it attempts an SQL query. Instead, use the aliasing feature of the IMS Enterprise Suite DLIModel utility plug-in to provide an alias for the restricted name. These keywords are not case-sensitive.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

The keywords shown in the following table are reserved SQL keywords.

ABORT to COT	CURRENT to IS	JOIN to REAL	REFERENCES to WORK
ABORT	CURRENT	JOIN	REFERENCES
ABS	CREATE	LAST	RESET
ACOS	CROSS	LEADING	REVOKE
ANALYZE	CURRENT	LEFT	RIGHT
AND	CURSOR	LIKE	ROLLBACK
ALL	DECIMAL	LISTEN	ROW
ALLOCATE	DECLARE	LN	ROWS
ALTER	DEFAULT	LOAD	SELECT
AND	DEGREES	LOCAL	SET
ANY	DELETE	LOCK	SETOF
ARE	DESC	LOG	SHOW
AS	DISTINCT	LOG10	SIGN
ASC	DO	MAX	SIN
ASSERTION	DOUBLE	MIN	SINH
ASIN	DROP	MOD	SMALLINT

ABORT to COT	CURRENT to IS	JOIN to REAL	REFERENCES to WORK
AT	END	MOVE	SQRT
ATAN	EXECUTE	NAMES	SUBSTRING
ATAN2	EXISTS	NATIONAL	SUM
AVG	EXP	NATURAL	TABLE
BEGIN	EXPLAIN	NCHAR	TAN
BETWEEN	EXTRACT	NEW	TANH
BINARY	EXTEND	NO	TO
BIT	FALSE	NONE	TRAILING
BOOLEAN	FETCH	NOT	TRANSACTION
BOTH	FIRST	NOTIFY	TRIM
BY	FLOAT	NULL	TRUE
CASCADE	FLOOR	NUMERIC	UNION
CAST	FOR	ON	UNIQUE
CEIL	FOREIGN	ONLY	UNLISTEN
CEILING	FROM	OR	UNTIL
CHAR	FULL	ORDER	UPDATE
CHARACTER	GRANT	OUTER	USER
CHECK	GROUP	PARTIAL	USING
CLOSE	HAVING	POSITION	VACCUM
CLUSTER	IN	POWER	VALUES
COLLATE	INNER	PRECISION	VARCHAR
COLUMN	INSERT	PRIMARY	VARYING
COMMIT	INT	PRIVILEGES	VERBOSE
CONSTRAINT	INTEGER	PROCEDURE	VIEW
COPY	INTERVAL	PUBLIC	WHERE
COS	INTO	RADIANS	WITH
COSH	IS	REAL	WORK
COT			

SQL statement usage with the IMS classic JDBC driver

The following information covers usage rules and restrictions that apply to SQL statements passed to IMS with the IMS classic JDBC driver provided in the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

SELECT statement usage:

The following information covers using the SQL SELECT statement with the IMS classic JDBC driver provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

IMS classic JDBC driver usage

The SELECT statement is used to retrieve data from a table. The result is returned in a tabular result set.

When using the SELECT statement with the IMS classic JDBC driver:

- All data in tables along the hierarchical path from the root table to the target table are implicitly included in the query results, and therefore they do not need to be explicitly stated.
- If the tables in the FROM clause do not share any columns with the same names, you can select columns without table-qualifying the column name.
- You do not need to qualify tables with the PCB name in the query unless the query is ambiguous without it.

INSERT statement usage:

The following information covers using the SQL INSERT statement with the IMS classic JDBC driver provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

IMS classic JDBC driver usage

An INSERT statement inserts a segment instance with the specified data under any number of parent segments that match the criteria specified in the WHERE clause.

The following figure shows an example of an INSERT statement that inserts a segment occurrence in the database using the DealershipDB PCB:

Sample INSERT statement

```
INSERT INTO DealershipDB.Sales (DateSold, PurchaserLastName,
    PurchaserFirstName, PurchaserAddress, SoldBy, StockVINNumber)
VALUES ('07032000', 'Beier', 'Otto', '101 W. 1st Street',
   'Springfield, OH', 'S123', '1ABCD23E4G5678901234')
WHERE Dealer.DealerNumber = 'A123'
AND Model.ModelTypeCode = 'K1'
```

One difference in syntax between JDBC queries to relational databases and to IMS is that standard SQL does not have a WHERE clause in an INSERT statement because tuples are being inserted into the table that is specified by the INTO keyword. In an IMS database, you are actually inserting a new instance of the specified segment, so you need to know where in the database this segment occurrence should be placed. With an INSERT statement, the WHERE clause is always necessary, unless you are inserting a root segment. With a prepared statement, the list of values can include a question mark (?) as the value that can be substituted before the statement is executed. For example:

```
INSERT INTO DealershipDB.Model(ModelTypeCode, CarMake, CarModel, CarYear, Price,
                               EPACityMileage, EPAHighwayMileage, Horsepower)
VALUES (?,?,?,?,?,?,?,?,?)
WHERE  Dealer.DealerNumber=?
```

DELETE statement usage:

The following information covers using the SQL DELETE statement with the IMS classic JDBC driver provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

IMS classic JDBC driver usage

A DELETE statement can delete any number of segment occurrences that match the criteria specified in the WHERE clause.

A DELETE statement with a WHERE clause also deletes the child segments of the matching segments. If no WHERE clause is specified, all of the segment occurrences of that type are deleted as are all of their child segment occurrences. The following code shows an example of a DELETE statement:

Sample DELETE statement

```
DELETE FROM DealershipDB.Order
WHERE Dealer.DealerNumber = '123' AND OrderNumber = '345'
```

UPDATE statement usage:

The following information covers using the SQL UPDATE statement with the IMS classic JDBC driver provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

IMS classic JDBC driver usage

An UPDATE statement modifies the value of the fields in any number of segment occurrences.

An UPDATE statement applies its SET operation to each instance of a specified segment with matching criteria in the WHERE clause. If the UPDATE statement does not have a WHERE clause, the SET operation is applied to all instances of the specified segment.

A SET clause contains at least one assignment. In each assignment, the values to the right of the equal sign are computed and assigned to columns to the left of the equal sign. For example, the UPDATE statement in the following code is called to accept an order. When a customer accepts an order, the Order segment's `SerialNo` and `DeliverDate fields` are updated.

Sample UPDATE statement

```
UPDATE DealershipDB.Order
SET SerialNo = '93234', DeliverDate = '12/11/2004'
WHERE OrderNumber = '123'
```

FROM clause usage:

The following information covers using the SQL FROM clause with the IMS classic JDBC driver provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

IMS classic JDBC driver usage

In a SQL SELECT statement, the FROM clause indicates which tables to use to retrieve the data.

When using the FROM clause with the IMS classic JDBC driver:

- Do not join segments in the FROM clause.
- List only one segment in the FROM clause.
- List the lowest-level segment that is used in the SELECT list and WHERE clause.
- Qualify the segment in the FROM clause by using the PCB name.

The behavior of a FROM clause in the IMS classic JDBC driver differs from standard SQL in that explicit joins are not required or allowed. Instead, the lowest-level segment in the query (in the SELECT statement and WHERE clause) must be the only segment that is listed in the FROM clause. The lowest-level segment in the FROM clause is equivalent to a join of all the segments, starting with the one that is listed in the FROM clause up the hierarchy to the root segment.

For example, in an IMS classic JDBC driver client application, theFROM clause `FROM pcb01.ILLNESS` is equivalent to the following FROM clause in a SQL join statement:

```
FROM pcb01.HOSPITAL, pcb01.WARD, pcb01.PATIENT, pcb01.ILLNESS,
```

PCB-qualified SQL queries

In Java applications using the IMS JDBC drivers, connections are made to PSBs. Because there are multiple database PCBs in a PSB, there must be a way to specify which PCB (using its alias) in a PSB to use when executing an SQL query on the java.sql.Connection object. To specify which PCB to use, always qualify segments that are referenced in the FROM clause of an SQL statement by prefixing the segment name with the PCB name. You can omit the PCB name only if the PSB contains only one PCB.

In the following SQL query example, the PCB name *pcb01* qualifies the HOSPITAL segment.

```
SELECT * FROM pcb01.HOSPITAL
```

WHERE clause usage:

In a SQL SELECT statement, the WHERE clause is used to select data conditionally.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

When using the WHERE clause with the IMS classic JDBC driver, use columns that are in any table from the root table down to the table in the FROM clause.

The IMS JDBC drivers convert the WHERE clause in an SQL query to an segment search argument (SSA) list when querying a database. SSA rules restrict the type of conditions you can specify in the WHERE clause. The following restrictions apply:

- Compare columns to values, not other columns. You can use the following operators between column names and values in the individual qualification statements:
  ```
  <
  <=
  =
  =<
  >
  !=
  ```
  For example, the following WHERE clause will fail because it is trying to compare two columns:
  ```
  WHERE PAYMENTS.PATNUM=PAYMENTS.AMOUNT
  ```
  The following example is valid because the WHERE clause is comparing a column to a value:
  ```
  WHERE PAYMENTS.PATNUM='A415'
  ```
- **Recommendation:** Qualify columns with table names. If you do not table-qualify a column, there can be ambiguity if that column exists in more than one table that was joined in the FROM clause.
- Do not use parentheses. Qualification statements are evaluated from left to right. The order of evaluation for operators is the IMS evaluation order for segment search arguments.
- List all qualification statements for a table adjacently. For example, in the following valid WHERE clause, the qualified columns from the same PATIENT table are listed adjacently:
  ```
  WHERE PATIENT.PATNAME='BOB' OR PATIENT.PATNUM='A342' AND WARD.WARDNO='M52'
  ```
  The following invalid WHERE clause will fail because the columns from the HOSPITAL table are separated by the columns from the WARD table:
  ```
  WHERE HOSPITAL.HOSPNAME='Santa Teresa' AND WARD.WARDNO='M52'
  OR WARD.WARDNAME='Cardiology' AND HOSPITAL.HOSPCODE='90'
  ```
- The OR operator can be used only between qualification statements that contain columns from the same table. You cannot use the OR operator across tables. To combine qualification statements for different tables, use an AND operator. For example, the following invalid WHERE clause will fail:
  ```
  WHERE WARD.WARDNO='D03' OR PATIENT.PATNUM='A415'
  ```
  However, the following WHERE clause is valid because the OR operator is between two qualification statements for the same table:
  ```
  WHERE PATIENT.PATNUM='A409' OR PATIENT.PATNAME='Sandy'
  ```
- The columns in the WHERE clause must be DBD-defined fields. These columns that are in the DBD are marked in the DLIModel IMS Java report as being either primary key fields or search fields.
- When using prepared statements, you can use the question mark (?) character, which is later filled in with a value. For example, the following WHERE clause is valid:
  ```
  WHERE PAYMENTS.AMOUNT>?
  ```

SQL aggregate functions supported by the IMS JDBC drivers

The IMS classic JDBC driver and the IMS Universal JDBC driver support SQL aggregate functions and related keywords.

- AS
- AVG
- COUNT

- GROUP BY
- MAX
- MIN
- ORDER BY
  - ASC
  - DESC
- SUM

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

Restriction: The supported SQL aggregate functions accept only a single field name in a segment as the argument (the DISTINCT keyword is not allowed). The exception is the COUNT function, which allows the DISTINCT keyword. This restriction applies to the IMS classic JDBC driver, but not the IMS Universal JDBC driver.

The ResultSet type for aggregate functions and ORDER BY and GROUP BY clauses is always TYPE_SCROLL_INSENSITIVE.

The following table shows the data types of the fields that are accepted by the aggregate functions, along with the resulting data type in the ResultSet.

Table 108. Supported SQL aggregate functions and their supported data types

Function	Argument type	Result type
SUM and AVG	Byte	Long
	Short	Long
	Integer	Long
	Long	Long
	BigDecimal	Double-precision floating point
	Single-precision floating point	Double-precision floating point
	Double-precision floating point	Double-precision floating point
MIN and MAX	Any type except BIT, BLOB, or BINARY	Same as argument type
COUNT	Any type	Long

Column names generated by aggregate functions

The ResultSet column name from an aggregate function is a combination of the aggregate function name and the field name separated by an underscore character (_). For example, the statement `SELECT MAX(age)` results in a column name MAX_age. Use this column name in all subsequent references—for example, `resultSet.getInt("MAX_age")`.

If the aggregate function argument field is table-qualified, the ResultSet column name is the combination of the aggregate function name, the table name, and the column name, separated by underscore characters (_). For example, `SELECT MAX(Employee.age)` results in a column name MAX_Employee_age.

Using the AS clause

You can use the AS keyword to rename the aggregate function column in the result set or any other field in the SELECT statement. You cannot use the AS keyword to rename a table in the FROM clause. When you use the AS keyword to rename the column, you must use this new name to refer to the column. For example, if you specify `SELECT MAX(age) AS oldest`, a subsequent reference to the aggregate function column is `resultSet.getInt("oldest")`.

If you are using the IMS Universal JDBC driver and you specified a SELECT query with column names renamed by an AS clause, you can only refer to the field in the resulting ResultSet by the AS rename. However, in the rest of your SELECT query, in the WHERE, ORDER BY, and GROUP BY clauses, you can use either the original column name or the AS rename.

Using the ORDER BY and GROUP BY clauses

Important: The field names that are specified in a GROUP BY or ORDER BY clause must match exactly the field name that is specified in the SELECT statement.

When using the IMS Universal JDBC driver, the following queries with the ORDER BY and GROUP BY clauses are valid:

```
SELECT HOSPNAME, COUNT(PATNAME) AS PatCount FROM PCB01.HOSPITAL, PATIENT
GROUP BY HOSPNAME ORDER BY HOSPNAME
```

```
SELECT HOSPNAME, COUNT(DISTINCT PATNAME) AS PatCount FROM PCB01.HOSPITAL,
PATIENT GROUP BY HOSPNAME ORDER BY HOSPNAME
```

Using the COUNT function with DISTINCT

When using the IMS Universal JDBC driver, the COUNT aggregate function can be qualified with the DISTINCT keyword. For example, the following query returns all hospital names listed in ascending order along with the number of distinct patient names from that hospital. The COUNT aggregate function generates a column name COUNT_DISTINCT_PATNAME .

```
SELECT HOSPNAME, COUNT(DISTINCT PATNAME)FROM PCB01.HOSPITAL, PATIENT GROUP
BY HOSPNAME ORDER BY HOSPNAME
```

PreparedStatement class of the IMS classic JDBC driver

The following information covers using the PreparedStatement class with the IMS classic JDBC driver provided by the classic Java APIs for IMS and the IMS Java dependent region resource adapter.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

To improve performance of an application that uses the IMS classic JDBC driver, use the `PreparedStatement` class of the IMS classic JDBC driver with SQL statements. The `PreparedStatement` class completes the initial steps in preparing queries only once so that you need to provide the parameters only before each repeated database call.

The `PreparedStatement` object performs the following actions only once before repeated database calls are made:

1. Parses the SQL.
2. Cross-references the SQL with the `DLIDatabaseView` object, found in the Java class libraries for IMS.
3. Builds SQL into SSAs before a database call is made.

Important: You must use a prepared statement when you store XML into a database.

Note: You can also use XQuery support with the IMS classic JDBC driver to further refine your query results.

IMS-specific usage for the IMS classic JDBC driver

The following information covers IMS-specific usage for the IMS classic JDBC driver provided by the classic Java APIs for IMS.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

Although the JDBC interface to an IMS database closely follows the relational database paradigm, the segments are physically stored in a hierarchical database, which affects the semantics of your JDBC calls to some extent. To avoid unexpected results or potential performance problems, follow these recommendations:

- When you code a SELECT list, generally try to supply predicates in the WHERE clause for all levels down the hierarchy to your target segment.

  If you supply a predicate in the WHERE clause for a target segment somewhere down the hierarchy and omit predicates for its parents, IMS must scan all candidate segments at the parent levels in an attempt to match the predicate that you supplied. For example, if you are retrieving a second-level segment and you supply a predicate for that second-level segment, but do not supply one for the root segment, IMS might perform a full database scan, testing every second-level segment under every root against the predicate. This has performance implications, particularly at the root level, and also might result in unexpected segments being retrieved. A similar consideration applies to locating segments for UPDATE clauses.
- When you insert a new segment, generally try to supply predicates in the WHERE clause for all levels down the hierarchy to your target new segment.

  If you omit a predicate for any level down to the insert target segment, IMS chooses the first occurrence of a segment at that level that allows it to satisfy remaining predicates, and performs the insert in that path. This might not be what you intended. For example, in a three-level database, if you insert a third-level segment, and supply a predicate for the root but none at the second level, your new segment will always be inserted under the first second-level segment under the specified root.
- If you delete a segment that is not a bottom-level (leaf) segment in its hierarchy, you also delete the remaining segments in that hierarchical subtree. The entire family of segments of all types that are located hierarchically below your target deleted segment are also typically deleted.
- When you provide predicates to identify a segment, the search is faster if the predicate is qualified on a primary or secondary index key field, rather than

simply on a search field. Primary and secondary key fields are identified for each segment in the DLIModel IMS Java report.

Sample application that uses the IMS classic JDBC driver

The following information covers a sample application that uses the IMS classic JDBC driver provided by the classic Java APIs for IMS and the IMS Java dependent region resource adapter.

Recommendation: Because the IMS Universal JDBC driver provides improved support for standard SQL syntax, use the IMS Universal JDBC driver to develop JDBC applications that access IMS.

Because IMS is a hierarchical database, the classic Java APIs for IMS do not fully implement the JDBC API.

To use the JDBC driver for IMS to read, update, insert, and delete segment instances, an application must:

1. Obtain a connection to the database. Load the DLIDriver and retrieve a Connection object from the DriverManager.
2. Retrieve a Statement or PreparedStatement object from the Connection object and execute it. An example of this step is in the code example below.
3. Iterate the ResultSet object returned from the Statement or PreparedStatement object to retrieve specific field results. An example of this step is in code example below.

The following code sample, which is part of a sample method showModelDetails, obtains a Connection object, retrieves a PreparedStatement object, makes SQL calls to the database, and then iterates the ResultSet object that is returned from the PreparedStatement object.

Example JDBC application

```
public ModelDetailsOutput showModelDetails(ModelDetailsInput input)
throws NamingException, SQLException, IMSException {

    // Extract the key from the input
    String modelKey = input.getModelKey();
    ModelDetailsOutput output = new ModelDetailsOutput();

    // Validate the key
    if (modelKey != null && !modelKey.trim().equals("")) {

        // Build the SQL query.
        String query = "SELECT * FROM Dealer.ModelSegment WHERE "
                + "ModelSegment.ModelKey = '" + input.getModelKey() + "'";

        // Execute the query
        Statement statement = connection.createStatement();
        ResultSet results = statement.executeQuery(query);

        // Store the results in the output object and send it
        // back to the caller of this method.
        if (results.next()) {
            output.setMake(results.getString("Make"));
            output.setModelType(results.getString("ModelType"));
            output.setModel(results.getString("Model"));
            output.setYear(results.getString("Year"));
            output.setPrice(results.getString("MSRP"));
            output.setCount(results.getString("Counter"));
```

```
            }
        }
        return output;
    }
```

Problem determination for Java applications

You can debug your Java applications to determine the source of problems within your applications.

Exceptions thrown from IMS DL/I calls

Exceptions are thrown as a result of non-blank status codes and non-zero return codes (in cases when there were no PCBs to deliver status codes) from IMS DL/I calls. Even though an exception is thrown by the `JavaToDLI` class for every non-blank status code, some of these exceptions are caught by the application or database packages and converted to return values.

How exceptions map to DL/I status codes

The `com.ibm.ims.base.IMSException` class extends the `java.lang.Exception` class.

The `DLIException` class extends the `IMSException` class. The `DLIException` class includes all errors that occur within the Java class libraries for IMS that are not a result of any call to IMS.

You can use the following methods to get information from an `IMSException` object:

getAIB
: Returns the IMS application interface block (AIB) from the DL/I call that caused the exception. The IMS AIB is null for the `DLIException` object. The methods on the AIB can be called to return other information at the time of the failure, including the resource or PCB name and the PCB itself.

getStatusCode
: Returns the IMS status code from the DL/I call that caused the exception. This method works with the `JavaToDLI` set of constants. The status code is zero (0) for a `DLIException` object.

getFunction
: Returns the IMS function from the DL/I call that caused the exception. The function is zero (0) for a `DLIException` object.

The following database access methods of the `DLIConnection` class return false if they receive a GB status code (no more such segments or segment not found) or a GE status code (no such segment or end of database):

- `DLIConnection.getUniqueSegment`
- `DLIConnection.getNextSegment`
- `DLIConnection.getUniqueRecord`
- `DLIConnection.getNextRecord`
- `DLIConnection.getNextSegmentInParent`

The `IMSMessageQueue.getUniqueMessage` method returns false if it receives a QC (no more messages) status code. The `IMSMessageQueue.getNextMessage` method returns false if it receives a QD status code, which means that there are no more segments for multi-segment messages.

The following code example extracts information from an IMSException object.

IMSException class example

```
try {
   DealerDatabaseView dealerView = new DealerDatabaseView();
   DLIConnection connection = DLIConnection.createInstance(dealerView);
   connection.getUniqueSegment(dealerSegment, dealerSSAList);
} catch (IMSException e ) {
   short statusCode = e.getStatusCode();
   String failingFunction = e.getFunction();
}
```

SQLException objects

An SQLException object is thrown to indicate that an error has occurred either in the Java address space or during database processing.

Each SQLException provides the following information:

- A string that describes the error.
  - This string is available through the use of the getMessage() method.
- An "SQLstate" string that follows XOPEN SQLstate conventions.
  - The values of the SQLstate string are described in the XOPEN SQL specification.
- A link to the next SQL exception if more than one was generated.
  - The next exception is used as a source of additional error information.

XML tracing for the classic Java APIs for IMS

Using the com.ibm.ims.base.XMLTrace class for z/OS applications or com.ibm.ims.rds.XMLTrace for distributed applications, you can debug your Java applications by tracing, or documenting, the flow of control throughout your application.

By setting up trace points throughout your application for output, you can isolate problem areas and, therefore, know where to make adjustments to produce the results you expect. In addition, because the XMLTrace class supports writing input parameters and results, and the methods within the classic Java APIs for IMS use this feature, you can verify that correct results occur across method boundaries.

The XMLTrace class replaces the IMSTrace class. However, applications that use the IMSTrace class will still function properly.

WebSphere Application Server security requirements for XML tracing

Before you can trace your application that runs on WebSphere Application Server for z/OS or WebSphere Application Server for distributed platforms, you must add permissions to the WebSphere Application Server server.policy file and create a was.policy for the application EAR file.

To add permissions to the WebSphere Application Server server.policy file:

1. Open the WebSphere Application Server server.policy file, which is in the properties directory of theWebSphere Application Server installation directory, and find the following code, which was added when you installed the custom service (if this code is not in the file, add it):

```
grant codeBase "file:/imsjava/-" {
      permission java.util.PropertyPermission "*", "read, write";
      permission java.lang.RuntimePermission "loadLibrary.JavTDLI";
      permission java.io.FilePermission "/tmp/*", "read, write";
};
```

2. Below `permission java.io.FilePermission "/tmp/*", "read, write";`, add the following permission, replacing `traceOutputDir` with the directory name for the trace output file:

   ```
   permission java.io.FilePermission "/traceOutputDir/*", "read, write";
   ```

To create the was.policy file:

1. Create a new file named was.policy that contains the following code, replacing `traceOutputDir` with the directory name for the trace output file:

   ```
   grant codeBase "file:${application}" {
     permission java.io.FilePermission "/traceOutputDir/*", "read, write";
   };
   ```

2. Put the was.policy file in the META-INF directory of your application's EAR file.

Enabling XML tracing

To debug with `XMLTrace`, you must first turn on the tracing function by calling one of the `XMLTrace.enable` methods.

Because tracing does not occur until this variable is set, it is best to do so within a static block of your main application class. Then, you must decide how closely you want to trace the your java application's flow of control and how much tracing you want to add to your application code.

You can determine the amount of tracing in your Java application by providing the trace level in the `XMLTrace.enable` method. By default, this value is set to `XMLTrace.TRACE_EXCEPTIONS`, which traces the construction of exceptions provided by the Java class libraries for IMS. `XMLTrace` also defines constants for three types of additional tracing. These constants provide successively more tracing from `IMSTrace.TRACE_CTOR1` (level-one tracing of constructions) to `IMSTrace.TRACE_DATA3` (level-three tracing of data).

`XMLTrace` has the following trace levels:

Trace level
Description

TRACE_EXCEPTIONS
Traces exceptions

TRACE_CTOR1
Traces level-1 constructors

TRACE_METHOD1
Traces level-1 parameters, return values, methods, and constructors

TRACE_DATA1
Traces level-1 parameters, return values, methods, and constructors

TRACE_CTOR2
Traces level-2 constructors

TRACE_METHOD2
Traces level-2 parameters, return values, methods, and constructors

TRACE_DATA2
Traces level-2 parameters, return values, methods, and constructors

TRACE_CTOR3
Traces level-3 constructors

TRACE_METHOD3
Traces level-3 parameters, return values, methods, and constructors

TRACE_DATA3
Traces level-3 parameters, return values, methods, and constructors

Tracing the methods of the Java class libraries for IMS:

You can programmatically enable tracing to so that system output can be redirected to a trace file.

To enable the tracing that is shipped with the methods of the Java class libraries for IMS:

1. Call the `XMLTrace.enable` method and specify the root element name and the trace level. For example:
   ```
   XMLTrace.enable("MyTrace", XMLTrace.TRACE_METHOD1);
   ```
2. Set an output stream (a print stream or a character output writer) as the current trace stream. For example:
   a. Set the system error stream as the current trace stream:
      ```
      XMLTrace.setOutputStream(System.err);
      ```
   b. Set a `StringWriter` object (or any other type of writer) as the current trace stream:
      ```
      StringWriter stringWriter = new StringWriter();
      XMLTrace.setOutputWriter(stringWriter);
      ```
3. Close the XML trace:
   ```
   XMLTrace.close();
   ```

Steps 1 and 2 are best implemented within a static block of your main application class, as shown in the following code example.

```
public static void main(String args[]){
    static {
        XMLTrace.enable("MyTrace", XMLTrace.TRACE_METHOD1);
        XMLTrace.setOutputStream(System.err);
    }
}
```

Tracing your application:

You can add trace statements to your application, similar to those provided by the Java class libraries for IMS, by defining an integer variable that you test prior to writing trace statements.

Using a variable other than `XMLTrace.libTraceLevel` enables you to control the level of tracing in your application independently of the tracing in the Java class libraries for IMS. For example, you can turn off the tracing of the routines of the Java class libraries for IMS by setting `XMLTrace.libTraceLevel` to zero, but still trace your application code.

To enable tracing for your application:

1. Define an integer variable to contain the trace level for application-provided code:`public int applicationTraceLevel = XMLTrace.TRACE_CTOR3;`
2. Set up the `XMLTrace` method to trace methods, parameters, and return values as necessary.

Enabling J2EE tracing in WebSphere Application Server

You can trace the IMS library classes by using the WebSphere Application Server tracing service.

You can also trace the IMS library classes or your applications using the `com.ibm.ims.base.XMLTrace` class. The `XMLTrace` class is class provided by the Java class libraries for IMS that represents the trace as an XML document. You can trace different levels of the code depending on the trace level.

Specifying the level of tracing:

To use the WebSphere Application Server for z/OS tracing service, you must first specify the level of tracing.

To specify the level of tracing:

1. In the left frame of the WebSphere Application Server for z/OS administrative console, click **Resources**, and then click **Resource Adapters**.

   A list of resource adapters is displayed.
2. Click the name of the IMS DB resource adapter.

   A configuration dialog is displayed.
3. Under Additional Properties, click **J2C connection factories**.

   A list of connection factories is displayed.
4. Click the name of the J2C connection factory for which you want to enable tracing.

   A configuration dialog is displayed.
5. Under Additional Properties, click **Custom Properties**.

   Properties are listed in a table.
6. Click **TraceLevel** row.
7. In the **Value** field, type the trace level.
8. Click **OK**.

   The properties table displays the trace level that you just entered.
9. In the messages box, click **Save**.

   The save page is displayed.
10. Click **Save** to update the master repository with your changes.

Specifying the application server and the package to trace:

After you specify the level of tracing, specify the application server and package to trace and then restart the server.

To specify the application server and the package to trace:

1. In the left frame of the WebSphere Application Server for z/OS, administrative console, click **Servers**, and then click **Application Servers**.

   A list of application servers is displayed.
2. Click the name of the server on which you want to enable tracing.
3. Under Troubleshooting, click **Diagnostic Trace Service**.

A configuration dialog for Diagnostic Trace Service is displayed.

4. Select the **Enable Log** check box and click **OK**.
5. Under Troubleshooting, click **Change Log Detail Levels**.
6. If you are using WebSphere Application Server for z/OS, click the plus sign (+) next to **com.ibm.connector2**.
7. If you are using WebSphere Application Server for distributed platforms, click the plus sign (+) next to **com.ibm.ims.***.
8. If you are using WebSphere Application Server for z/OS, click **com.ibm.connector2.ims.***.
9. If you are usingWebSphere Application Server for distributed platforms, click **com.ibm.ims.rds.***.
10. From the list of trace detail levels, click **all**.
11. If you are using WebSphere Application Server for z/OS, verify that `com.ibm.connector2.ims.*=all` appears in the text box and click **OK**.
12. If you are using WebSphere Application Server for distributed platforms, verify that `com.ibm.ims.rds.*=all` appears in the text box and click **OK**.
13. In the messages box, click **Save**.

    The save page is displayed.
14. Click **Save** to update the master repository with your changes.
15. Restart the server.

Specifying at runtime the application server and the package to trace:

You can turn tracing on and off by specifying at runtime the server and package to trace. You do not need to restart your server each time.

To specify the application server and the package to trace at runtime:

1. In the left frame of the WebSphere Application Server for z/OS administrative console, click **Servers**, and then click **Application Servers**.

   A list of application servers is displayed.
2. Click the name of the server on which you want to enable tracing.
3. Under Troubleshooting, click **Change Log Detail Levels**.

   A configuration dialog for Change Log Detail Levels is displayed.
4. Click the **Runtime** tab.
5. If you are using WebSphere Application Server for z/OS , click the plus sign (+) next to **com.ibm.connector2**.
6. If you are using WebSphere Application Server for distributed platforms, click the plus sign (+) next to **com.ibm.ims.rds**.
7. If you are using WebSphere Application Server for z/OS , click **com.ibm.connector2.ims.***
8. If you are using WebSphere Application Server for distributed platforms, click **com.ibm.ims.rds.***
9. From the list of trace detail levels, click **all**.
10. If you are using WebSphere Application Server for z/OS , verify that `com.ibm.connector2.ims.*=all` appears in the text box and click **OK**.
11. If you are using WebSphere Application Server for distributed platforms, verify that `com.ibm.ims.rds.*=all` appears in the text box and click **OK**.
12. Click **OK**.

Part 6. Appendixes

© Copyright IBM Corp. 1974, 2015

Notices

This information was developed for products and services offered in the U.S.A.

This material may be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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, MD-NC119
Armonk, NY 10504-1785
United States of America

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 to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

© Copyright IBM Corp. 1974, 2015

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS," without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

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 http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Programming interface information

This information documents Product-sensitive Programming Interface and Associated Guidance Information provided by IMS, as well as Diagnosis, Modification or Tuning Information provided by IMS.

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a section or topic, or by a Product-sensitive programming interface label. IBM requires that the preceding statement, and any statement in this information that refers to the preceding statement, be included in any whole or partial copy made of the information described by such a statement.

Diagnosis, Modification or Tuning information is provided to help you diagnose, modify, or tune IMS. Do not use this Diagnosis, Modification or Tuning information as a programming interface.

Diagnosis, Modification or Tuning Information is identified where it occurs, either by an introductory statement to a section or topic, or by the following marking: Diagnosis, Modification or Tuning Information.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, See IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Bibliography

This bibliography lists all of the publications in the IMS Version 11 library, supplemental publications, publication collections, and accessibility titles cited in the IMS Version 11 library.

IMS Version 11 library

Title	Acronym	Order number
IMS Version 11 Application Programming	APG	SC19-2428
IMS Version 11 Application Programming APIs	APR	SC19-2429
IMS Version 11 Commands, Volume 1: IMS Commands A-M	CR1	SC19-2430
IMS Version 11 Commands, Volume 2: IMS Commands N-V	CR2	SC19-2431
IMS Version 11 Commands, Volume 3: IMS Component and z/OS Commands	CR3	SC19-2432
IMS Version 11 Communications and Connections	CCG	SC19-2433
IMS Version 11 Database Administration	DAG	SC19-2434
IMS Version 11 Database Utilities	DUR	SC19-2435
IMS Version 11 Diagnosis	DGR	GC19-2436
IMS Version 11 Exit Routines	ERR	SC19-2437
IMS Version 11 Installation	INS	GC19-2438
IMS Version 11 Licensed Program Specifications	LPS	GC19-2439
IMS Version 11 Master Index and Glossary	MIG	SC19-2440
IMS Version 11 Messages and Codes, Volume 1: DFS Messages	MC1	GC19-4232
IMS Version 11 Messages and Codes, Volume 2: Non-DFS Messages	MC2	GC19-4233
IMS Version 11 Messages and Codes, Volume 3: IMS Abend Codes	MC3	GC19-4234
IMS Version 11 Messages and Codes, Volume 4: IMS Component Codes	MC4	GC19-4235
IMS Version 11 Operations and Automation	OAG	SC19-2441
IMS Version 11 Release Planning	RPG	GC19-2442
IMS Version 11 System Administration	SAG	SC19-2443
IMS Version 11 System Definition	SDG	GC19-2444
IMS Version 11 System Programming APIs	SPR	SC19-2445
IMS Version 11 System Utilities	SUR	SC19-2446

Supplementary publications

Title	Order number
IMS and SOA Executive Overview	GC19-2516
Program Directory for Information Management System Transaction and Database Servers V11.0	GI10-8788
IMS Version 11 Fact Sheet	GC19-2451
IRLM Messages and Codes	GC19-2666

© Copyright IBM Corp. 1974, 2015

Publication collections

Title	Format	Order number
IMS Version 11 Product Kit	CD	SK5T-7375

Accessibility titles cited in the IMS Version 11 library

Title	Order number
z/OS TSO/E Primer	SA22-7787
z/OS TSO/E User's Guide	SA22-7794
z/OS ISPF User's Guide Volume 1	SC34-4822

Index

Numerics

274X
  defining to operate with MFS 470
3270P Printer
  defining to operate with MFS 470
3290 Display Panel
  defining to operate with MFS 470
3601 workstation
  defining to operate with MFS 470
3770 Data Communication System
  defining to operate with MFS 470
3790 Communication System
  defining to operate with MFS 470
6670 Printer
  defining to operate with MFS 470

A

abend codes
  pseudo- 53
  S201 347
  U0069 55
  U0711 122
  U0777 46
  U1008 50
  U119 122
  U2478 46
  U2479 46
  U261 347
  U3301 50
  U3303 46
  U476 347
  U711 104
access methods
  DEDB 78
  description 73
  GSAM 80
  HDAM 75
  HIDAM 76
  HISAM 79
  HSAM 79
  MSDB 77
  PHDAM 73, 75
  PHIDAM 73, 76
  SHISAM 80
  SHSAM 80
access of
  IMS databases through z/OS 80
  segments through different paths 86
accessibility
  features xii
  keyboard shortcuts xii
accessing databases with application programs 36
accessing GSAM databases 305
addressability to UIB, establishing 237
addressing mode (AMODE) 252, 391
Advanced Function Printing (AFP) 449
AFPDS and IMS Spool API 449
aggregates, data 22
AIB (application interface block)
  address return 246
  AIB identifier (AIBID) 383
  AIBERRXT (reason code) 232
    AIB mask 383
  AIBID (AIB identifier) field, AIB mask 232
  AIBLEN (DFSAIB allocated length) field 383
  AIBLEN (DFSAIB allocated length) field, AIB mask 232
  AIBOALEN (maximum output area length) 232
    AIB mask 383
  AIBOAUSE (used output area length)
    AIB mask 383
    description 232
  AIBREASN (reason code) 232
  AIBRSA1 (resource address) 232
    AIB mask 383
  AIBRSNM1 (resource name)
    AIB mask 383
    description 232
  AIBSFUNC (subfunction code)
    AIB mask 383
    description 232
  defining storage 245, 386
  description 245, 386
  DFSAIB allocated length (AIBLEN) 232, 383
  fields 232, 383
  mask 232, 383
  program entry statement 246
  specifying 232, 383
AIB (Application Interface Block)
  supported commands 503
AIB identifier (AIBID)
  description 232
AIB interface 12
AIB mask, specifying 232
AIBERRXT (reason code) 232
AIBOALEN (maximum output area length) field, AIB mask 232
AIBOAUSE (used output area length) field, AIB mask 232
AIBREASN (reason code) AIB mask, field 232
AIBREASN (reason code) field, AIB mask 232
AIBRSA1 (resource address) field, AIB mask 232
AIBRSNM1 (resource name) field, AIB mask 232
AIBSFUNC (subfunction code) field, AIB mask 232
AIBTDLI interface 55, 245, 386
AJ status code 527
AL_LEN call 338, 342
allocation, dynamic 56
alternate destinations
  sending messages 409
alternate PCB 509
  alternate destinations 409
  change call 409
  CHNG call
    description 409
    using PURG with 409
  defining in ISRT call 408
  destination of modifiable alternate PCBs 409
  express 408
  modifiable
    description 408
    use 409
  modifiable alternate PCBs
    changing the destination 409
    CHNG call 409
    description 409
  modifiable PCBs 409
  PURG call
    description 409
    using CHNG with 409
  replying to one alternate terminal 409
  response 422
  SAMETRM=YES 422
  sending messages
    to several alternate destinations 409
    using alternate PCBs 409
    using the PURG call 409
  sending messages to other terminals 409
  types and uses 383
  use with program-to-program message switching 411
  using the CHNG call with 409
alternate PCB mask
  description 383
  format 383
alternate PCBs 107
alternate response PCBs 107
alternate terminals
  responding 409
AM status code 527
AMODE 252
AMODE(31) 501
analysis of
  processing requirements 35
  required application data 17
  user requirements 15
anchor point, root 75
AND operators
  dependent 294
  independent 294
API (application programming interface) for LU 6.2 devices
  explicit API 121
  implicit API 121
APPC 347
  application program types for LU 6.2 devices 111

© Copyright IBM Corp. 1974, 2015

APPC *(continued)*
basic conversation 114
description 111
entering IMS transactions from LU 6.2 devices 111
LU 6.2 partner program design
DFSAPPC message switch 144
flow diagrams 122
integrity after conversation completion 142
mapped conversation 114
RRS 347
APPC conversational program
CPI-C driven 430
ending the conversation 428
message switching 425
modified application program
MSC 429
remote execution, MSC 429
modified IMS application 429
application data
analyzing required 17
identifying 17
application design
analyzing
processing requirements 57
the data a program must access 58
user requirements 15, 17
data dictionary, using 20
DataAtlas 20
DB/DC Data Dictionary 20
debugging 449
designing a local view 22
documenting 15, 177
IMS Spool API interface 447
overview 15
application interface block (AIB) 232
Application Interface Block (AIB)
supported commands 503
application program 116
design
tasks overview 13
documentation 177
HALDB environment, scheduling 252
hierarchy examples 5
I/O areas, specifying 239, 385
sync point 275
test 147
TSO 46
application programming interface (API) 121
application programs
accessing databases 36
assembler language 213, 365
DL/I call formats 213, 365
CICS
testing 169
DL/I calls 213, 365
sample call formats 213, 365
Pascal 221, 373
PL/I 224, 376
application programs, IFP 509
applications
development tasks overview 13
APSB (allocate program specification block) 121, 347
area, I/O 12
AREALIST call 343
assembler language
DL/I call-level sample 194
DL/I command-level sample 484
DL/I program structure 192
entry statement 246
MPP coding 439
program entry 246
register 1, program entry 246
return statement 246
skeleton MPP 439
SSA definition examples 241
UIB, specifying 237
UIBDLTR
introduction 237
UIBFCTR
introduction 237
UIBPCBAL
introduction 237
asynchronous conversation, description for LU 6.2 transactions 113
AUTH call 99
authorization
ID DB2 for z/OS 99
security 99
availability enhancements, data 535
availability of data 10, 53, 69
sensitivity 70

B

back-out database changes 67
backing out
database changes 514
backout point 431
description 431
intermediate 514
intermediate (SETS/SETU) 435
backout point, intermediate 288
backout, dynamic 39
bank account database example 5
basic checkpoint 49, 65, 68
basic conversation, APPC 114
Basic edit
IMS TM 461
Basic Edit
input message 399
output message 399
translation to uppercase 399
basic edit, overview of 102
Batch Backout utility 39
batch environment 36
batch message processing (BMP) programs
issuing checkpoints 513
PCBs 509
batch message processing program. 42
batch programs
assembler language 192
C language 196
COBOL 199
command-level samples
assembler 484
C 495
batch programs *(continued)*
command-level samples *(continued)*
COBOL 488
PL/I 491
converting to BMPs 42
databases that can be accessed 36, 58
DB batch processing 39
differences from online 39
issuing checkpoints 50, 65, 513
maintaining integrity 284
Pascal 205
PL/I 207
recovery 39, 65
recovery of database 67
Batch Terminal Simulator (BTS) 148
batch-oriented BMPs. 42
BILLING segment 5
binder options with EXEC DLI 501
binding, reference 210
BKO execution parameter 39
block descriptor word (BDW)
IMS Spool API 449
BMP (batch message processing) program 42
batch-oriented 42
checkpoints in 50, 65
databases that can be accessed 42
description of 42
limiting number of locks with LOCKMAX= parameter 50
recovery 42
databases that can be accessed 36, 58
transaction-oriented
checkpoints in 50
databases that can be accessed 44
recovery 44
BMP (batch message processing) programs
basic checkpoint
issuing 513
checkpoint (CHKP)
description 513
EXEC DLI command 513
checkpoint (CHKP) command
in a Batch or BMP Program 513
issuing 513
checkpoint (CHKP) EXEC DLI command
current position 513
CHKP (Checkpoint)
description 513
EXEC DLI command 513
CHKP (Checkpoint) command
issuing in a Batch or BMP Program 513
CHKP (Checkpoint) EXEC DLI command
current position 513
committing your program's changes to a database 513
EXEC DLI recovery commands
CHKP (Checkpoint) 513
SYMCHKP (Symbolic Checkpoint) 513
I/O area
symbolic CHKP 513
issuing checkpoints 513

BMP (batch message processing)
programs *(continued)*
PCBs 509
planning for database recovery
CHKP command checkpoints 513
taking checkpoints 513
recovery EXEC DLI commands
basic CHKP 513
SYMCHKP 513
SYMCHKP (Symbolic Checkpoint)
command
description 513
BMPs, transaction-oriented
ROLB 284
Boolean operators
dependent AND 294
independent AND 294
SSA, coding 240
BTS (Batch Terminal Simulator) 148
buffer pool, STAT call and OSAM 152
buffer subpool, statistics for debugging
enhanced STAT call and
OSAM 156
VSAM 154, 161

C

C language
__pcblist 246
application programming
IMS database 216
IMS Transaction Manager 368
batch program, coding 196
DL/I call formats 216, 368
DL/I calls
sample call formats 216, 368
DL/I program structure 196
entry statement 246
exit 246
I/O area 216, 368
PCBs, passing 246
return statement 246
skeleton MPP 439
SSA definition examples 241
system function 246
C program
DL/I command-level sample 495
C/MVS 55
CALL statement (DL/I test
program) 147
call-level programs
comparing with command-level
programs
command codes and options 533
commands and calls 531
DL/I calls available to IMS and CICS
command-level 531
call-level programs, scheduling a PSB 62
callout request
asynchronous callout
programming model 478
asynchronous callout request 473
comparison of synchronous and
asynchronous callout requests 473
IMS TM Resource Adapter
Enterprise JavaBeans (EJB) 474
Java EE application 474
callout request *(continued)*
IMS TM Resource Adapter *(continued)*
message-driven bean (MDB) 474
web service 474
JMS (Java Message Service)
implementation
IMSQueueConnectionFactory 660
overview 473
resume tpipe
IRM_TIMER field 475
protocol 475
security 475
security exit routine
(DFSYRTUX) 475
RESUME TPIPE call 474
SOAP Gateway
web service 474
synchronous callout
COBOL code example 475
JBP (Java batch processing)
regions 660
JMP (Java message processing)
regions 660
programming model 475
synchronous callout request 473
synchronous program switch
request 473
user-written IMS Connect TCP/IP
application 474
calls, DL/I 12
calls, system service
calls,system service
SETS/SETU (set a backout
point) 288
PLITDLI 288
SETS (Set a Backout Point) call
description 288
SETS/SETU (set a backout point)
backing out to an intermediate
backout point 288
SETU (Set a Backout Point
Unconditional) call
description 288
SETU, call function 288
CCTL (coordinator controller)
restrictions
with BTS (Batch Terminal
Simulator) 148
with DL/I test program 147
CEETDLI
address return 246
program entry statement 246
checkpoint 65
basic 49, 65
calls, when to use 50
frequency, specifying 52, 65
IDs 65
in batch programs 50, 65
in batch-oriented BMPs 50, 65
in MPPs 50
in transaction-oriented BMPs 50
issuing 39
printing log records 65
restart 50, 68
summary of 49
symbolic 49, 65
checkpoint (CHKP) calls
considerations 191
description 283
issuing 283
checkpoints
relationship to commit point and sync
point 275
CHKP (checkpoint) 49
CHKP (checkpoint) call
considerations 191
CHKP (symbolic checkpoint) call
with GSAM 312
CHKPT=EOV 49
CHNG call
usage 452
with directed routing 415
CHNG system service call 447
CICS 12
command language translator 501
distributed transactions
accessing IMs 63
CICS applications
unit testing 169
CICS applicationsIMS batch regions
database logging 71
ESTAE routine 71
resource cleanup 71
STAE routine 71
CICS DL/I call program
compiling 192
CICS online programs
assembler language
sample 194
COBOL, sample 201
PL/I, sample 210
CIMS 347
classes schedule, example 29
classic Java APIs for IMS
CICS support 676
programming model 677
DB2 for z/OS stored procedures
support 675
programming model 676
deployment descriptor
requirements 674
IMS classic JDBC driver 678
resource adapters
IMS DB distributed resource
adapter 669
IMS DB resource adapter 669
transaction demarcation
java.sql.Connection interface 671
javax.transaction.UserTransaction
interface 670
WebSphere Application Server for
distributed platforms support 669
tracing 703, 704
WebSphere Application Server for
z/OS support 669, 674
IMS DB resource adapter 669
restrictions 673
tracing 703, 704
closing a GSAM database explicitly 309
CMPAT option 509
CMPAT=YES PSB specification 39
COBOL 55
application programming 219, 371

COBOL *(continued)*
copybook types 614, 684
data types
mapped to COBOL 614, 684
DL/I call formats 219, 371
DL/I call-level, sample 201
DL/I command-level sample 488
DL/I program structure 199
entry statement 246
mapping to IMS 614, 684
return statement 246
skeleton MPP 441
SSA definition examples 241
types
data, mapped to COBOL 614, 684
UIB, specifying 237
code
course 17
transaction 41
codes abend 46
codes, status
logical relationships 299
coding DC calls and data areas 438
in assembler language 439
in C Language 439
in COBOL 441
in Pascal 442
in PL/I 444
skeleton MPP 439, 441, 442, 444
coding rules, SSA 240
columns
fields, compared to 543
relational representation, in 543
command codes
D
examples 189
DEDBs 189
F
restrictions 337
overview 189
Q 291
qualified SSAs 189
restrictions 240
subset pointers 189
unqualified SSAs 189
command language translator, CICS 501
command-level program
DFHEIENT 484
DFHEIRET 484
DFHEISTG 484
parameters
EIBREG 484
RCREG 484
RCREG parameters 484
reentry 484
SCHD PSB command 484
command-level programs
adjustable character string 507
array, connected 507
assembler language
I/O area 507
automatic storage 491
C code standard header file 495
character string
adjustable 507
fixed-length 507
command-level programs *(continued)*
COBOL
I/O area 507
commands
SCHD PSB 491
comparing with call-level programs
command codes and options 533
commands and calls 531
concatenated key, segment 507
connected array 507
DIB (DL/I interface block) 503
DL/I calls available to IMS and CICS 531
EIBREG parameter 484
fixed-length character string 507
GE status code 491, 495
I/O area
assembler language 507
COBOL 507
coding 507
PL/I 507
restriction 507
I/O area, defining 507
key feedback area, defining 507
major structure 507
minor structure 507
PL/I
I/O area 507
preparing EXEC DL/I program for execution 501
reentrance 484
restrictions
I/O area 507
I/O area, PL/I 507
samples
assembler language 484
C 495
COBOL 488
PL/I 491
SCHD PSB command 491
segment
concatenated key 507
standard header file, C code 495
status codes
GE 491, 495
structure
major 507
minor 507
commands, EXEC DLI 12
COMMENTS statement 147
commit 276
single-phase 279
UOR 278
commit point 431
process 275
relationship to check point and sync point 275
commit point processing
DEDB 336
MSDB 327
commit points 39, 46, 65
communicating with other IMS TM systems 413
COMPARE statement 147
comparing EXEC DLI
commands with DL/I calls 531
options with command codes 533
comparison of symbolic CHKP and basic CHKP 49
comparison to ROLB and ROLS call 431
comparison to ROLL and ROLB call 431
comparison to ROLL and ROLS call 431
compiler, COBOL 488
compiling, options with EXEC DLI 501
concatenated data sets, GSAM 316
concatenated segments, logical relationships 297
concurrent access to full-function databases 39
Connection object 698
considerations in screen design 102
CONTINUE-WITH-TERMINATION indicator 55
continuing a conversation 104
control, passing processing 12
conventions, naming 15
conversation attributes
asynchronous 113
MSC synchronous and asynchronous 113
synchronous 113
conversation state, rules for APPC verbs 115
conversational mode
description 107
LU 6.2 transactions 113
conversational processing
abnormal termination of, precautions for 105
coding necessary information 428
deferred program switch 104
designing a conversation 104
DFSCONE0 105
example 417
for APPC/IMS 427
gathering requirements 104
how to continue the conversation 104
how to end the conversation 104
immediate program switch 104
overview 104, 416
passing control and continuing the conversation 423
passing the conversation to another program 104
recovery considerations 105
replying to the terminal 422
SPA 105
structure 418
use with alternate response PCBs 107
using a deferred program switch to end the conversation 104
using ROLB, ROLL and ROLS in 422
what happens in a conversation 104
conversational program
definition 416
conversations, preventing abnormal termination 105
converting an existing application 15
coordinator controller. 147
coordinator, sync-point 116
copybook types 614, 684
course code 17

CPI Communications driven program
sync point 275
CPI Resource Recovery calls 275
creation of
a new hierarchy 86
reports 20
crossing a unit of work (UOW) boundary
when processing DEDBs 530
currency of data 3
current position
determining 257
current roster 17

D

data
a program's view 10
aggregate 22
documentation 20
elements, homonym 19
elements, isolating repeating 23
elements, naming 19
hierarchical relationships 5
integrity, how DL/I protects 64
keys 23
recording its availability 20
relationships, analyzing 22
structuring 22
unique identifier 19
data areas
coding 191
data availability
considerations 53, 69
levels 10
recording 20
data availability enhancements 535
data capture 245, 386
data currency 3
data definition 15
data dictionary
DataAtlas 20, 177
DB/DC Data Dictionary 20, 177
documentation for other
programmers 177
in application design 20
data element
description 17
homonym 19
isolating repeating 23
listing 17
naming 19
synonym 19
data elements, grouping into
hierarchies 23
data entity 17
data entry database 36
data mask 12
data propagation
sync point 280
data redundancy 3
data redundancy, reducing 297
data sensitivity 10
data sensitivity, defined 91
data storage methods
combined file 3
in a database 3
separate files 3
data structure 10
data structure conflicts, resolving 81
data structures 191
DataAtlas 20, 177
database
access to 58
administrator 5
availability
obtaining information 535
status codes, accepting 535
calls
Fast Path 345
changes, backing out 67
DBCTL facilities
REFRESH command 535
description (DBD) 10
example, medical hierarchy 5
hierarchy 5
integrity, maintaining 513
options 73
planning for recovery
backing out database changes 514
position
determining 257
record, processing 12
recovery 513
recovery with ROLL call 284
recovery, back out changes 284
REFRESH command 535
unavailability 53, 69
database and data communications
security 15
DATABASE macro 95
database record 5
database recovery
backing out 432
backing out database changes 67
checkpoints, description 65
planning
XRST command 514
restarting your program,
description 68
Database Resource Adapter (DRA) 347
database statistics, retrieving 151
database types
areas 36
DB2 for z/OS 36, 58
DEDB 36, 78
description 36
full-function 36
GSAM 36, 80
HDAM 75, 76
HISAM 79
HSAM 79
MSDB 36, 77
PHDAM 73, 75
PHIDAM 73, 76
relational 36
root-segment-only 36
SHISAM 80
SHSAM 80
databases
accessing with application
programs 36
DB batch processing 39
DB Control
DRA (Database Resource
Adapter) 347
DB Control DRA (Database Resource
Adapter) 347
DB PCB
definition 509
DB PCB (database program
communication block) 10
concatenated key and PCB mask 306
database
DB PCB, name 306
database name 306
entry statement, pointer 306
fields 230
fields in a DB PCB 306
key feedback area 306
length field in DB PCB 306
key feedback area length field 306
length of key feedback area 306
mask
fields 306
fields, GSAM 306
general description 230
name 306
relation 230
specifying 230
masks
DB PCB 230
multiple DB PCBs 273
processing options
field in DB PCB 306
processing options field 306
relation to DB PCB 230
RSA (record search argument)
overview 306
secondary indexing, contents 296
status code field 306
status codes
field in DB PCB 306
undefined-length records 306
variable-length records 306
DB/DC
Data Dictionary 20, 177
environment 36
DB2 (DATABASE 2)
with IMS TM 407, 445
DB2 for z/OS
databases 36, 58
DB2 for z/OS access
application programming 667
committing work 667
drivers 667
IMS databases, compared to 667
rolling back work 667
DBA 5
DBASF, formatted OSAM buffer pool
statistics 152
DBASS, formatted summary of OSAM
buffer pool statistics 152
DBASU, unformatted OSAM buffer pool
statistics 152
DBCTL
environment 36
DBCTL (Database Control)
single-phase commit 279
two-phase commit 276

DBCTL environment 58
DBCTL facilities
ACCEPT command 535
data availability 535
QUERY command 535
ROLS (Roll Back to SETS or SETU) command 514
SETS (Set a Backout Point) command 514
DBCTLID parameter 347
DBD (database description) 10
DBD (database description), sample 678
DBESF, formatted OSAM subpool statistics 156
DBESO, formatted OSAM pool online statistics 156
DBESS, formatted summary of OSAM pool statistics 156
DBESU, unformatted OSAM subpool statistics 156
DCCTL
environment 36
DDM (distributed data management) 550
deadlock, program 39
dealership samples
DBD 678
PSB 678
debug a program, How to 174
debugging
XMLTrace 700
DEDB
DL/I calls
AL_LEN call 342
AREALIST call 343
DEDBINFO call 344
DEDSTR call 345
DI_LEN call 343
DS_LEN call 343
summary 338
DEDB (data entry database) 78, 322
call restrictions 337
crossing a unit of work (UOW) boundary when processing 530
data entry database 517
database
processing, Fast Path 517
dependent segments
sequential 517
direct dependent segments, in DEDBs 517
DL/I calls 337
Fast Path
database, processing 517
multiple qualification statements 188
processing
commit point 336
DEDBs 517
fast path 319
H option 337
overview 517
P option 336
POS (Position) command 528
POS call 333
subset pointers 328, 517
segment
sequential dependent 517
DEDB (data entry database) *(continued)*
sequential dependent segments in DEDBs 517
updating segments 323
updating with subset pointers 328
DEDB (data entry database) and the PROCOPT operand 93
DEDBINFO call 344
DEDSTR call 345
deferred program switch 104
defining application program elements to IMS
AIB 503
AIB (Application Interface Block)
AIB mask 503
restrictions 503
Application Interface Block (AIB)
AIB mask 503
restrictions 503
DIB 503
execution diagnostic facility 503
I/O area 507
key feedback area 507
restrictions
AIB 503
Transaction Server, CICS 503
defining subset pointers 520
definition
data 15
dependent segment 5
root segment 5
Delete (DLET) call
with MSDB, DEDB or VSO DEDB 323
DELETE keyword 691
example 691
dependent AND operator 294
dependent segment 5
dependent segments
retrieving 86
sequential
identfying free space 529
locating a specific dependent 528
locating the last inserted dependent 529
dependents
direct 36
sequential 36
dependents, direct 322
description, segment 5
design efficiency, programs 483
design of
an application 15
conversation 104
local view 22
designing
terminal screen 102
determination of mappings 28
device input format (DIF), control block 102
device output format (DOF), control block 102
devices supported by MFS 470
DFSAPPC 425
DFSAPPC
format 425
option keywords 425
DFSAPPC *(continued)*
message switching 425
DFSAPPC message switch 144
DFSCONE0 (Conversational Abnormal Termination exit routine) 105
DFSDDLT0 (DL/I test program) 147
DFSDLTR0 (DL/I image capture). 171
DFSHALDB ddname
selective partition processing 301
DFSLI000 (language interface module)
binding COBOL code to 201
DFSMDA macro 56
DI_LEN call 343
diagnosing multiple parsing error return codes 452
DIB (DL/I interface block)
accessing information 503
assembler language program
DIB fields 503
variable names, mandatory 503
BA status code 503
BC status code 503
C program
DIB fields 503
variable names, mandatory 503
CICS
HANDLE ABEND command 503
COBOL program
DIB fields 503
variable names, mandatory 503
FH status code 503
fields 503
FW status code 503
GA status code 503
GB status code 503
GD status code 503
GE status code 503
GG status code 503
GK status code 503
II status code 503
label restriction 503
labels 503
LB status code 503
NI status code 503
PL/I
program variable names, mandatory 503
restrictions
DIB label 503
status codes
BA 503
BC 503
FH 503
FW 503
GA 503
GB 503
GD 503
GE 503
GG 503
GK 503
II 503
LB 503
NI 503
TG 503
structure 503
translator version 503
DIB (DLI interface block) 12

dictionary, data 20
DIF (device input format), control block 102
direct access methods
  characteristics 74
  HDAM 75
  HIDAM 76
  PHDAM 73, 75
  PHIDAM 73, 76
  types of 74
direct dependents 36
directed routing 413
distributed data management (DDM) 550
distributed presentation management 472
Distributed Relational Database Access (DRDA) 549, 550
distributed relational database architecture (DRDA)
  DDM commands 360
  overview 359
Distributed Sync Point 120
DL/I
  calls
    for CICS and IMS programs 531
DL/I access methods
  considerations in choosing 73
  DEDB 78
  direct access 74
  GSAM 80
  HDAM 75
  HIDAM 76
  HISAM 79
  HSAM 79
  MSDB 77
  PHDAM 73, 75
  PHIDAM 73, 76
  sequential access 78
  SHISAM 80
  SHSAM 80
DL/I call trace 147
DL/I calls 12, 378
  general information
    coding 191
  image capture
    batch job 150
  log data set
    DFSERA50 call trace exit routine 151
  relationships to PCB types
    I/O PCBs 378
  sample call formats 219, 221, 371, 373
  tracing
    DLITRACE control statement 150
    image capture 149, 150, 151, 354
    IMS TRACE command 354
    TRACE command 150
DL/I calls (general information)
  qualification statements
    overview 182
  qualified calls 182
  qualified SSAs (segment search arguments)
    structure 182
DL/I calls (general information) (continued)
  qualifying calls
    command codes 189
    field 182
    segment type 182
  segment search arguments (SSAs) 182
  SSAs (segment search arguments)
    qualified 182
  unqualified calls 182
DL/I calls, system service
  ROLB 284
  ROLL 284
DL/I calls, testing DL/I call sequences 147, 171
DL/I database
  access to 58
  description 58
DL/I image capture (DFSDLTR0) programs 171
DL/I interface block 503
DL/I language interfaces 213, 365
  overview 213, 365
  supported interfaces 213, 365
DL/I options
  field level sensitivity 81
  logical relationships 86, 297
  secondary indexing 82, 293
DL/I program ROLB scenario 122
DL/I test program (DFSDDLT0)
  call statements 147
  checking program performance 147
  comments statements 147
  compare statements 147
  control statements 147
  description 147
  status statements 147
  testing DL/I call sequences 147, 171
DLET (Delete) call
  with MSDB, DEDB or VSO DEDB 323
DLIConnection class 699
DLIDriver
  loading 698
  registering 686
DLIModel IMS Java report 678
DLIModel utility plug-in
  DLIModel database report 678
DLITPLI 248
documentation for users 177
documentation of
  data 20
  the application design process 15
DOF (device output format), control block 102
DPM (distributed presentation management)
  using 472
  with ISC 472
DRA (Database Resource Adapter)
  description 347
  startup table 347
DRDA (Distributed Relational Database Access) 549, 550
DRDA (distributed relational database architecture)
  DDM commands 360
  overview 359
driver
  registering with DriverManager 686
DriverManager facility 686
DS_LEN call 343
duplicate values, isolating 23
dynamic allocation 56, 71
dynamic backout 39, 514
dynamic MSDBs (main storage databases) 5

E

EBCDIC 65
editing
  considerations in your application 102
  messages
    considerations in message and screen design 102
    overview 101
editing messages
  edit routines
    Basic Edit 398
    Intersystem Communication (ISC) Edit 398
    Message Format Service (MFS) 398
efficient program design 483
elements
  data, description 17
  data, naming 19
emergency restart 449
EMH (expedited message handler) 41
end a conversation, how to 104
enhanced STAT call formats for statistics
  OSAM buffer subpool 156
  VSAM buffer subpool 161
entity, data 17
entry and return conventions 246
entry point
  AIB (application interface block)
    address return 387
    and program entry statement 387
  assembler language
    program entry 387
    register 1 at program entry 387
  C language
    __pcblist 387
    entry statement 387
    exit 387
    longjmp 387
    passing PCBs 387
    return 387
    system function 387
  CEETDLI
    address return 387
    program entry statement 387
  COBOL
    DLITCBL 387
  entry point
    assembler language 387
  overview 387

entry point *(continued)*
Pascal
entry statement 387
passing PCBs 387
PL/I
passing PCBs 387
pointers in entry statement 387
environments
DB/DC 36
DBCTL 36
DCCTL 36
options in 36, 58
program and database types 36
equal-to relational operator 182
ERASE parameter 93
error
execution 174
initialization 174
ESTAE routines 55
example
current roster 17
field level sensitivity 81
instructor schedules 29
instructor skills report 29
local view 29
logical relationships 86
schedule of classes 29
examples
bank account database 5
Boolean operators 187
conversational processing 417
D command code 189
FLD/CHANGE 326
FLD/VERIFY 326
medical database 5
multiple qualification statements 187
path call 189
UIB, defining 237
exceptions
description 699
EXEC DLI 520
binder options, required 501
compiler options, required 501
DLI option 501
preparing program for execution 501
PROCESS statement overrides 501
recovery commands
XRST (Extended Restart) 514
translator options, required 501
z/OS & VM 501
z/OS & VM translator 501
EXEC DLI commands 12
EXEC DLI program translating 501
execution errors 174
existing application, converting an 15
explicit API for LU 6.2 devices 121
explicitly opening and closing a GSAM database 309
express alternate PCB 408
express PCBs 107
Extended Restart 49, 68

F

F command code
restrictions 337
Fast Path 36
Fast Path *(continued)*
database calls 319, 320
databases 36
databases, processing 319
DEDB (data entry database) 78
processing 319
DEDB and the PROCOPT operand 93
IFPs 41
MSDB (main storage database) 36, 77
processing 319
P (position) processing option 530
subset pointers with DEDBs 517
subset pointers, using with DEDBs 328
types of databases 319
field
changing contents 325
checking contents: FLD/VERIFY 323
Field (FLD) call 323
field level sensitivity
as a security mechanism 91
defining 10
description 81
example 81
specifying 81
uses 81
field name
FSA 324
SSA
qualification statement 182
field search argument (FSA)
description 323
with DL/I calls 323
field value
FSA 325
SSA qualification statement 182
fields
columns, compared to 543
default value 691
in SQL queries 543
File Select and Formatting Print Program (DFSERA10) 49
FIN (Finance Communication System)
defining to operate with MFS 470
fixed, MSDBs (main storage databases) 5
FLD (Field) call
description 323
FLD/CHANGE 325
FLD/VERIFY 323
flow diagrams, LU 6.2 122
CPI-C driven commit scenario 122
DFSAPPC, synchronous SL=none 122
DL/I program backout scenario 122
DL/I program commit scenario 122
local CPI communications driven program, SL=none 122
local IMS Command
asynchronous SL=confirm 122
local IMS command, SL=none 122
local IMS conversational transaction, SL=none 122
local IMS transaction
asynchronous SL=confirm 122
asynchronous SL=none 122
flow diagrams, LU 6.2 *(continued)*
local IMS transaction *(continued)*
synchronous SL=confirm 122
synchronous SL=none 122
multiple transactions in same commit 122
remote MSC conversation
asynchronous SL=confirm 122
asynchronous SL=none 122
synchronous SL=confirm 122
synchronous SL=none 122
formats
PSB 509
frequency, checkpoint 52
FSA (field search argument)
description 323
with DL/I calls 323
full-function databases
and the PROCOPT operand 93
how accessed, CICS 58
how accessed, IMS 36

G

gather requirements
for conversational processing 104
gathering requirements
for database options 73
for message processing options 99
GC status code 530
general programming guidelines 483
Generalized Sequential Access Method (GSAM) 80
program access 305
generalized sequential access method (GSAM))
DB PCB (database program communication block)
mask 245
getFunction method 699
getNextException 700
GO processing option 50
GPSB (generated program specification block)
format 389
greater-than relational operator 182
greater-than-or-equal-to relational operator 182
group data elements
into hierarchies 23
with their keys 23
GSAM (generalized sequential access method
GSAM (generalized sequential access method)
RSA 308
record search argument 308
GSAM (generalized sequential access method)
accessing databases 305
BMP region type 317
call summary 312
CHKP 312
coding considerations 312
data areas 245
data set
attributes, specifying 316

GSAM (generalized sequential access method) *(continued)*
data set *(continued)*
characteristics, origin 313
concatenated 316
DD statement DISP parameter 314
extended checkpoint restart 315
database, explicitly opening and closing 309
DB PCB masks 245
DBB region type 317
description 305
designing a program 305
DLI region types 317
fixed-length records 310
I/O areas 310
record formats 310
records, retrieving and inserting 308
restrictions on CHKP and XRST 312
RSA 245
RSA (record search argument) description 308
status codes 311
undefined-length records 310
variable-length records 310
XRST 312
GSAM (Generalized Sequential Access Method)
accessing GSAM databases 58
database type 36
description 80
GSAM PCB 509
guidelines, general programming 483
guidelines, programming 181

H

H processing option 337
HALDB
DFSHALDB ddname 301
HALDB control statement 301
partitions
selective partition processing 301
selective partition processing 301
HALDB (High Availability Large Database) 83
application programs
scheduling against 252
initial load 252
HALDB control statement 301
HDAM
multiple qualification statements 188
HDAM (Hierarchical Direct Access Method) 75
HIDAM (Hierarchical Indexed Direct Access Method) 76
hierarchical database
example 543
relational database, compared to 543
hierarchical database example, medical 5
Hierarchical Direct Access Method (HDAM) 75
Hierarchical Indexed Direct Access Method (HIDAM) 76
Hierarchical Indexed Sequential Access Method (HISAM) 79
Hierarchical Sequential Access Method (HSAM) 79
hierarchy
bank account database 5
data structures 191
description 5
grouping data elements 23
medical database 5
hierarchy examples 5
High Availability Large Database (HALDB) 83
application programs
scheduling against 252
initial load 252
HISAM (Hierarchical Indexed Sequential Access Method) 79
homonym, data element 19
Hospital database example 557
HOUSHOLD segment 5
HSAM (Hierarchical Sequential Access Method) 79

I

I/O area 12
command-level program 507
specifying 239, 385
XRST 514
I/O PCB 509
in different environments 60
mask
12-byte time stamp 226
general description 226
group name field 226
input message sequence number 226
logical terminal name field 226
message output descriptor name 226
specifying 226
status code field 226
userid field 226
userid indicator field 226
I/O PCB mask
general description 379
specifying 379
IBM Enterprise COBOL for z/OS
Java dependent region interoperability 665
Java dependent regions
backend application for Java applications 666
frontend application for Java applications 667
issuing DL/I calls in COBOL 666
identification of
recovery requirements 50
identifying
application data 17
online security requirements 99
output message destinations 107
security requirements 91
IDs, checkpoint 65
IFP (IMS Fast Path) program
databases that can be accessed 36
IFP (IMS Fast Path) program *(continued)*
differences from an MPP 41
recovery 41
restrictions 41
IFP application programs 509
ILLNESS segment 5
image capture program
CICS application program 171
IMS application program 148
immediate program switch 104
implicit API for LU 6.2 devices 121
IMS application
diagnosing
abnormal termination (abend) 165
program execution errors 165
program initialization errors 165
IMS application programs, standard 428
IMS classic JDBC driver
IMS-specific usage 697
IMS conversations
conversational program 416
nonconversational program 416
IMS database
database design
logical relationships 89
IMS Explorer for Development
com.ibm.ims.db.DLIDatabaseView class
generating 557
DLIDatabaseView class
generating 557
Java metadata class
generating 557
IMS Fast Path (IFP) programs, description of 41
IMS Java dependent region resource adapter
Java batch processing (JBP) regions 642
Java message processing (JMP) regions 642
IMS solutions for Java development
overview 539
IMS Spool API
dynamic allocation
error messages 456
print data sets
CHNG call 456
z/OS services for Dynamic Output (SVC109) 456
IMS Spool API application design 447
IMS support for DRDA 549
IMS TM
application program
message Type 393
DB2 considerations 407, 445
IMS Universal Database resource adapter 549
CCI programming interface 570
configuring SSL support
container-managed environment 638
WebSphere Application Server 638
connectivity
RRSLocalOption 563

IMS Universal Database resource adapter *(continued)*
connectivity *(continued)*
type-2 563
type-4 563
creating a CCI Connection
managed environment 563
creating a CCI ConnectionFactory
managed environment 563
DLIInteractionSpec class
deleting data 572
inserting data 572
retrieving data 572
updating data 572
JNDI lookup
connecting to IMS 563
logging 639
overview 561
sample application 570
specifying IMSConnectionSpec properties
managed environment 563
SQLInteractionSpec class
deleting data 576
inserting data 576
retrieving data 576
updating data 576
tracing 639
transaction management
bean managed 561
container managed 561
local transaction support 561
LocalTransaction interface 561
UserTransaction interface 561
XA transaction support 561
WebSphere Application Server for distributed platforms support 563
WebSphere Application Server for z/OS support 563
IMS Universal DL/I driver 549
adding segments
example 630
AIB
example 634
AIB interface 618
application programming 615
batchDelete
example 634
batchRetrieve 622
example 624
batchUpdate
example 632
com.ibm.ims.base 616
com.ibm.ims.dli 616
commit
example 635
configuring SSL support
stand-alone environment 638
connecting
IMS database 616
connections
IMS database 616
properties 616
create
example 630
creating segments
example 630
IMS Universal DL/I driver *(continued)*
DBPCB
example 634
DBPCB interface 618
delete
example 632
deleting multiple segments
example 634
deleting segments
batch 634
example 632, 634
DL/I DLET 632
DL/I ISRT 630
DL/I REPL 631
fetch size 624
setting 626
getNext
example 622
getNextWithinParent 622
getPathForBatchUpdate
example 632
getPathForInsert
example 630
getPathForRetrieveReplace
example 622, 631, 632
getUnique
example 622
GSAMPCB interface 618
IMSConnectionSpec
creating 616
example 616
IMSConnectionSpecFactory
example 616
IMSStatusCodes 634
insert
example 630
inserting segments
example 630
interfaces 618
Java packages 616
logging 639
overview 615
Path
data transformation 610, 626, 681
retrieving java.sql data types 610, 626, 681
Path interface 618
PathSet interface 618
PCB interface 618
programming model 615
PSB
creating 616
example 616
PSB interface 618
PSBFactory
example 616
query performance
improving 626
replace
example 631
retrieving
error code extension 634
reason code 634
return code 634
status code 634
retrieving data
example 622
IMS Universal DL/I driver *(continued)*
retrieving multiple segments
example 624
retrieving segments
batch 624
example 622, 624
rollback
example 635
segment search arguments
specifying 619
setFetchSize 626
SSAList
adding intitial qualification 619
appending additional qualifications 619
creating 619
debugging 619
qualified 619
setting command codes 619
setting lock classes 619
unqualified 619
SSAList interface 618
SSAs
specifying 619
tracing 639
transactions
example 635
local 635
one-phase commit 635
processing 635
scope 635
unit of recovery 635
unit of work 635
updating multiple segments
example 632
updating segments
batch 632
example 631, 632
IMS Universal drivers
application platforms 553
architecture 550
CICS support 550
com.ibm.ims.db.DLIDatabaseView class
generating 557
configuring SSL support
container-managed environment 638
stand-alone environment 638
WebSphere Application Server 638
connectivity
distributed (type-4) 550
local (type-2) 550
data access methods 553
DB2 for z/OS stored procedures support 550
DLIDatabaseView class
generating 557
Java dependent region support 550
Java metadata class
generating 557
JBP region support 550
JMP region support 550
overview 549
programming approaches 553

IMS Universal drivers *(continued)*
SSL support
container-managed environment 638
stand-alone environment 638
transaction processing options 553
variable length database segments 555
WebSphere Application Server for distributed platforms support 550
WebSphere Application Server for z/OS support 550
IMS Universal JCA/JDBC driver
connecting 567
data operations
DELETE 579
INSERT 579
PreparedStatement 579
SELECT 579
Statement 579
syntax 579
UPDATE 579
deploying 567
IMS Universal JDBC driver 549
columns compared to fields 592
configuring SSL support
stand-alone environment 638
connecting to IMS
DataSource 583
DriverManager 587
foreign key fields 597
example 597
SQL statement usage 597
hierarchical databases compared to relational databases 592
IMS-specific SQL usage
AS clause 598
DELETE statement 602
DISTINCT clause 598
FROM clause 598
GROUP BY clause 598
INSERT statement 600
ORDER BY clause 598
SELECT statement 598
UPDATE statement 601
WHERE clause 603
interfaces
DataSource 583
DriverManager 587
JDBC programming interface 591
logging 639
rows compared to segment instances 592
sample application 591
supported drivers 582
tables compared to segments 592
tracing 639
IMSException object
getStatusCode method 699
in-doubt UOR
definition 278
in-flight UOR
definition 278
independent AND operator 294
indexed field in SSA 293
indexing, secondary
DL/I Returns 296
indexing, secondary *(continued)*
effect on program 293
multiple qualification statements 294
status codes 296
INIT system service call 53
initialization errors 174
input for a DL/I program 191
input message
format 394
MFS 401
INQY system service call 53
INSERT keyword 691
example 691
WHERE clause 691
inserting a segment
GSAM records 308
instructor
schedules 29
skills report 29
integrity
batch programs 284
how DL/I protects data 64
maintaining,database 284
read without 95
using ROLB 284
MPPs and transaction-oriented BMPs 284
using ROLL 284
using ROLS 284
interface, AIB 12
intermediate backout point
backing out 288
intermediate backout points 514
Introduction to Resource Recovery 116
invalid processing and ROLB/SETS/ROLLS calls 105
IPDS and IMS Spool API 449
ISC (intersystem communication)
defining to operate with MFS 472
ISC (Intersystem Communication) 41
isolation of
duplicate values 23
repeating data elements 23
ISRT (Insert) call
with MSDB, DEDB or VSO DEDB 323
ISRT call
issuing to other terminals 408
message call
in conversational programs 418
referencing alternate PCBs 408
usage 408
ISRT system service call 447
issue checkpoints 39
issuing
checkpoints in batch or BMP programs 513

J

Java Batch Processing (JBP)
applications 46
databases that can be accessed 36
Java batch processing (JBP) application
accessing GSAM data 656
program switching
immediate 662
Java batch processing (JBP) application *(continued)*
programming models 652
restart 652
symbolic checkpoint 652
Java batch processing (JBP) regions
DB2 for z/OS access
application programming 667
IMS Java dependent region resource adapter 642
JBP applications 641
overview 641
Java class libraries for IMS 699
DL/I status codes
mapping to exceptions 699
DLIException class 699
exceptions
mapping to DL/I status codes 699
getStatusCode method 699
IMSException class 699
IMSException object
getAIB method 699
getFunction method 699
IMSMessageQueue 699
JDBC application 698
problem determination 699
status codes
mapping 699
Java Database Connectivity (JDBC) 549
Java EE Connector Architecture (JCA) 549
Java Message Processing (JMP)
applications 45
databases that can be accessed 36
Java message processing (JMP) application
DB2 for z/OS data access 645
IMS data access 645
IMSFieldMessage class
subclassing 643
input messages
defining 643
message handling 647
input messages 646
output messages
defining 643
program switching
deferred 664
immediate 662
programming models 645
transactions
commit 647
rollback 647
Java message processing (JMP) applications
message handling
conversational transactions 647
multi-segment messages 649
multiple input messages 651
repeating structures 650
scratchpad area (SPA) 647
Java message processing (JMP) regions
DB2 for z/OS access
application programming 667
IMS Java dependent region resource adapter 642

Java message processing (JMP) regions *(continued)*
JMP applications 641
overview 641
Java metadata class
IMS classic JDBC driver 557
IMS Explorer for Development generating 557
IMS Universal drivers 557, 606
java.sql.Connection interface 686
java.sql.DatabaseMetaData interface 686
java.sql.Driver interface 686
java.sql.PreparedStatement interface 686
java.sql.ResultSet interface 687
java.sql.ResultSetMetaData interface 686
java.sql.Statement interface 686
JBP (Java Batch Processing)
applications 46
databases that can be accessed 36
JBP (Java batch processing) regions
DB2 for z/OS access application programming 667
IMS Java dependent region resource adapter 642
synchronous callout support 660
JCA (Java EE Connector Architecture) 549
JDBC
BIGINT 610, 681
BINARY 610, 681
BIT 610, 681
CHAR 610, 681
classic Java APIs
portable SQL keywords restrictions 689
CLOB 610, 681
connecting to IMS database 686
Connection object, returning 686
DATE 610, 681
DOUBLE 610, 681
FLOAT 610, 681
IMS-specific SQL usage
FROM clause 692
SELECT statement 690
WHERE clause 693
WHERE clause subfield support 604
INTEGER 610, 681
interfaces
java.sql.Connection 686
java.sql.DatabaseMetaData 686
java.sql.Driver 686
java.sql.PreparedStatement 686
java.SQL.ResultSet 687
java.sql.ResultSetMetaData 686
java.sql.Statement 686
interfaces, limitations 686
jdbc:dli 686
mapping SQL data types to Java 610, 681
overview 581
PACKEDDECIMAL 610, 681
ResultSet
data transformation 610, 626, 681
retrieving java.sql data types 610, 626, 681
sample application 698
JDBC *(continued)*
SMALLINT 610, 681
SQL aggregate function
AS 593, 694
ASC 593, 694
AVG 593, 694
COUNT 593, 694
DESC 593, 694
GROUP BY 593, 694
MAX 593, 694
MIN 593, 694
ORDER BY 593, 694
SUM 593, 694
SQL aggregate functions supported 593, 694
SQL keywords supported 593, 688
TIME 610, 681
TIMESTAMP 610, 681
TINYINT 610, 681
Universal drivers
portable SQL keywords restrictions 595
using 698
writing an application 698
XML support
Java metadata class 606
overview 605
retrieval 608
SQL INSERT 606
SQL SELECT 608
storage 606
type-4 connectivity 605
ZONEDDECIMAL 610, 681
JDBC (Java Database Connectivity) 549
JES Spool/Print server 449
JMP (Java Message Processing)
applications 45
databases that can be accessed 36
JMP (Java message processing) regions
DB2 for z/OS access application programming 667
IMS Java dependent region resource adapter 642
synchronous callout support 660
JOURNAL parameter 448

K

key feedback area
command-level program 507
key sensitivity 91
keyboard shortcuts xii
keys, data 23

L

Language Environment
characteristics of CEETDLI 251, 390
LANG= Option on PSBGEN for PL/I Compatibility with Language Environment 251, 390
Language Environment
LANG = option for PL/I compatibility 251, 390
supported languages 251, 390
Language Environment, with IMS 251, 390
Large Data Sets 316
legal notices
notices 707
trademarks 707
less-than relational operator 182
less-than-or-equal-to relational operator 182
limit access with signon security 99
link editing, EXEC DLI 501
link to another online program 62
LIST parameter 149
listing data elements 17
LL field
in input message 394
in output message 395
local view
designing 22
examples 29
locating
a specific sequential dependent 528
last inserted sequential dependent 529
locating dependents in DEDBs
last-inserted sequential dependent, POS call 333
POS call 333
specific sequential dependent, POS call 333
lock management 291
locking protocol 93
LOCKMAX= parameter, BMP programs 50
log
records
sync points 279
LOG call
description 165
use in monitoring 174
log records
type 18 65
X'18' 49
LOG system service call 355
log, system 39
logical child 297
logical parent 297
logical relationships
defining 86
description 86
effect on programming 299
example 86
introduction 297
logical child 297
logical parent 297
physical parent 297
processing segments 297
programming, effect 297
status codes 299
logical structure 297
LTERM, local and remote 144
LU 6.2
conversations 427
support for APPC 111
LU 6.2 devices, signon security 99
LU 6.2 partner program design
DFSAPPC message switch 144

LU 6.2 partner program design *(continued)*
flow diagrams 122
integrity after conversation completion 142
scenarios 122
LU 6.2 User Edit Exit
using 407

M

macros
DATABASE 95
DFSMDA 56
TRANSACT 44
main storage database (MSDB) 77
main storage database (MSDBs)
types
nonrelated 5
main storage databases (MSDBs)
dynamic 5
types
related 5
maintaining database integrity 513
managing subset pointers in DEDBs with command codes 320
many-to-many mapping 28
mapped conversation, APPC 114
mappings, determining 28
mask
AIB 232
DB PCB 230
mask, data 12
master terminal
issuing timeout 413
medical database example 5
description 5
segments 5
message 393
editing
description 398
input message 399, 401
output 399
output message 407
skipping line 399
using Basic Edit 399
using ISC Edit 400
using LU 6.2 User Edit Exit 407
using MFS Edit 400
from terminals 393
I/O PCB 398
input 394, 401
input descriptor (MID), control block 102
input fields
contents 394
ISC (intersystem communication)
editing output messages 400
ISC (intersystem communication) edit
output message 400
message formatting service 399
MFS (Message Format Service)
editing message 399
output 107, 395, 407
output descriptor (MOD), control block 102
message *(continued)*
output fields
contents 395
printing 399
processing of 393
summary 396
processing options 99
receiving by program 393
result 398
sending to other application programs 411
type
message switch service 393
types 393
another terminal 393
Message Format Service 461
Message Format Service (MFS)
control blocks
relationship with screen format 467
LU 6.2 device restriction 401
secondary logical unit (SLU) 401
terminal
message processing program (MPP) 401
Message Input
Segment Format 394
message processing options
sending message to originating terminal 107
message processing program 439
message-driven programs
definition 437
supported message destinations 437
usage restrictions 437
methods of data storage
combined file 3
database 3
separate files 3
MFS (Message Format Service)
components 469
control blocks 102
relationship with screen format 467
editing output messages 400
example 466
input message
formats 401
MFS (message format service)
message editor 469
online performance 461
output message
formats 407
overview 102, 461
pool manager 469
remote programs 470
supported devices 470
MFS control blocks
DIF (device input format)
description 462
DOF (device output format)
description 462
MID (message input descriptor)
description 462
MOD (message output descriptor)
description 462
summary 462
MFS libraries
online change function 469
MFSTEST procedure (language utility)
pool manager 469
MID (message input descriptor), control block 102
mixed-language programming 252, 391
MOD (message output descriptor), control block 102
mode
multiple 50
processing 46
response 107
single 50
MODE parameter 46
MOVENEXT option
examples 520
use when moving subset pointer forward 520
moving subset pointer forward 520
MPP (message processing program)
coding in assembler language 439
coding in C language 439
coding in COBOL 441
coding in Pascal 442
coding in PL/I 444
coding necessary information 438
databases that can be accessed 36, 41
description 41
executing 41
input 438
parmcount 444
PL/I
entry statement restrictions 444
MPP coding notes 444
optimizing compiler 444
MPPs 509
ROLB 284
MSC (multiple systems coupling)
description 413
directed routing 413
receiving messages from other IMS TM systems 414
sending messages to other IMS TM systems 415
MSDB (main storage database) 36, 77
call restrictions 321
commit point processing 327
updating segments 323
MSDBs (main storage database)
processing commit points 327
MSDBs (main storage databases)
nonrelated 321
terminal related 321
types
description 321
nonrelated 5
related 5
multiple
DB PCBs 273
processing 267
qualification statements 186
DEDB 188
HDAM 188
PHDAM 188
multiple mode 46, 50

multiple positioning
advantages of 270
effecting your program 271
resetting position 273
multiple systems coupling 413
MVS SJF (Scheduler JCL Facility) 448
MYLTERM 321

N

names of data elements 19
naming conventions 15
NDM (Non-Discardable Messages) routine 46
network-qualified LU name 144
nonconversational program
definition 416
nonrelated (non-terminal-related) MSDBs 321
NOSTAE and NOSPIE 55
not-equal-to relational operator 182
NTO (Network Terminal Option) 470

O

ODBA 347
application execution environment
establishing 347, 351
application programs
testing 352
writing 347
CIMS 347
DB2 for z/OS stored procedures 351
DRA (Database Resource Adapter) 347
RRS 347
server program 350
ODBA (Open Database Access) 347
one-to-many mapping 28
online performance 461
online processing
databases that can be accessed 58
description 60
linking and passing control to other applications 62
performance, maximizing 63
online programs 41
online programs, command-level samples
assembler 484
C 495
COBOL 488
PL/I 491
online security
password security 99
supplying information about your application 99
terminal 99
Open Database Access (ODBA) program
application interface block (AIB)
fields 234
Open Database Access (ODBA) programs
abnormal termination (abend)
diagnosing 356
initialization errors 356
running errors 356
Open Database Access (ODBA) programs *(continued)*
application interface block (AIB)AERTDLI interface 251
tracing
DFSDDLT0 354
image capture 353, 354
operator
FSA 324
SSA 182
operators
AND operators
logical 186
Boolean 186
Boolean operators
logical AND operator 186
logical OR 186
OR operators
logical 186
relational 186
relational operators
independent AND 186
logical AND 186
logical OR 186
options
CMPAT 509
MOVENEXT 520
P processing 530
options for subset pointers
MOVENEXT 520
OSAM buffer pool, retrieving statistics 152
OTMA, processing conversations with 430
output message
format 395
printing 399
sending 411
to other application programs 411
to other IMS TM systems 415
using Basic Edit 399
using MFS 407
with directed routing 415
output messages, identifying destinations for 107
overlap, storage 507

P

P processing option 336, 530
parameters
BKO 39
DBCTLID 347
ERASE 93
JOURNAL 448
LIST 149
LOCKMAX 50
MODE 46
PROCOPT 93
TRANSACT 46
TXTU 448
WFI 44
parsing error return codes 452
Partitioned Hierarchical Direct Access Method (PHDAM) 73, 75
Partitioned Hierarchical Indexed Direct Access Method (PHIDAM) 73, 76
Partitioned Secondary Index (PSINDEX) 83
partitions
DFSHALDB ddname 301
HALDB control statement 301
selective partition processing 301
Pascal
application programming 221, 373
batch program, coding 205
DL/I call formats 221, 373
DL/I program structure 205
entry statement 246
PCBs, passing 246
skeleton MPP 442
SSA definition examples 241
syntax diagram, DL/I call format 221
pass control of processing 12
pass control to other applications 62
passing control
to a conversational program 423
to another program in a conversation 423
password security 99
path call
definition 189
example 189
overview 189
PATIENT segment 5
PAYMENT segment 5
PCB (program communication block)
12-byte time stamp, field in I/O PCB 226
address list, accessing 237
alternate 509
call 62
description 10
express 107
group name, field in I/O PCB 226
GSAM (generalized sequential access method)
DB PCB mask, fields 306
I/O PCB mask
12-byte time stamp 379
group name field 379
input message sequence number 379
logical terminal name field 379
message output descriptor name 379
status code field 379
userid field 379
in application programs, summary 509
input message sequence number, field in I/O PCB 226
logical terminal name, field in I/O PCB 226
masks
GSAM databases 306
I/O PCB 226, 379
message output descriptor name, field in I/O PCB 226
modifiable alternate PCBs 283
PCB (program communication block) types 509
RACF signon security 226, 379

PCB (program communication block) *(continued)*
RACROUTE SAF 226
signon security, RACF 226
status codes, field in I/O PCB 226
types 389
userid, field in I/O PCB 226
PCB lists 389
PCB parameter list in assembler language MPPs 439
PCB, express alternate 408
performance
impact 448
maximizing online 63
PHDAM
multiple qualification statements 188
PHDAM (Partitioned Hierarchical Direct Access Method) 73, 75
PHIDAM (Partitioned Hierarchical Indexed Direct Access Method) 73, 76
physical parent 297
physical structure of a database 10
PL/I
application programming
DL/I call formats 224, 376
DL/I calls 224, 376
batch program, coding 207
DL/I call-level sample 210
DL/I command-level sample 491
DL/I program, multitasking restriction 207
entry statement 246
PCBs, passing 246
pointers in entry statement 246
return statement 246
skeleton MPP 444
UIB, specifying 237
PL/I language 55
PLITDLI procedure
description 255
pool manager
MFS 469
POS (Position) call
description 333
POS (Position) command
identifying free space 529
locating a specific sequential dependent 528
locating the last inserted sequential dependent 529
using with DEDBs 528
Position (POS) command
identifying free space 529
locating a specific sequential dependent 528
locating the last inserted sequential dependent 529
using with DEDBs 528
position, reestablishing with checkpoint calls 50, 65
positioning
after DLET 259
after ISRT 261
after REPL 261
after retrieval calls 259
after unsuccessful DLET or REPL call 263
positioning *(continued)*
after unsuccessful retrieval or ISRT call 263
CHKP, effect
modifiable alternate PCBs 283
current position
unsuccessful calls 263
database position
unsuccessful calls 263
determining 257
not-found status code
description 263
position after 263
positioning
after unsuccessful calls 263
understanding current 257
preloaded programs 252
prepared statements
java.sql.PreparedStatement interface 686
PreparedStatement object 698
preparing programs
for EXEC DLI 483
for EXEC DLI execution 501
primarily sequential processing 79
print checkpoint log records, how to 65
problem determination 174
Problem Determination 699
procedures
CBLTDLI 255
PLITDLI 255
process database records 12
process of requests 12
process of requirements, analyzing 35, 57
processing
commit-point in DEDB 336
commit-point in MSDB 327
current position
multiple positioning 267
database position
multiple positioning 267
database, several views 273
DEDBs 328
Fast Path
P (position) option 530
Fast Path databases 319
GSAM databases 305
multiple
positioning 267
options
H (position), for Fast Path 337
P (position), for Fast Path 336
segments in logical relationships 297
single positioning 267
processing a message 396
processing mode 46
processing options
A (all) 93
D (delete) 93
defined 91
E (exclusive) 93
G (get)
description and concurrent record access 93
general description 93
processing options *(continued)*
GO (read only)
description 93
invalid pointers and T and N options 93
N option 93
risks of GOx options 93
T option 93
I (insert) 93
K (key) 91
R (replace) 93
PROCOPT parameter 93
PROCOPT=GO 50
program
design 191
design efficiency 483
restarting 284
program communication block 230
I/O PCB mask
userid indicator field 379
userid indicator, field in I/O PCB 226
program communication block (PCB) 10
program deadlock 39
program restarting
EXEC DLI XRST command 514
program sensitivity 53
program specification block (PSB) 10
program structure
conversational 418
conversational processing
message formats 418
restrictions 418
ROLB call 418
ROLL call 418
ROLS call 418
steps in a conversational program 418
deferred program switch
passing control to another 418
immediate program switch 418
LL field 418
message
in conversations 418
ROLB call
use in conversations 418
ROLL call
use in conversations 418
ROLS call
use in conversations 418
SPA (scratchpad area)
contents 418
format 418
inserting 418
restrictions on using 418
saving information 418
system service calls
ROLB call 418
ROLL call 418
ROLS call 418
program switch
deferred 104
immediate 104
program test 147
program types, environments and database types 36
program waits 50

program-to-program message switching
conversational 423
conversational processing
by deferred switch 423
by immediate switch 423
ending the conversation and passing control 423
passing control and continuing the conversation 423
restrictions 423
deferred program switch
in conversational programs 423
ending a conversation and passing control to another program 423
immediate program switch
in conversational programs 423
MSC (multiple systems coupling)
conversational programming 423
nonconversational 411
passing a conversation to another IMS TM system 423
passing control
restrictions 423
restrictions 411
security checks 411
SPA (scratchpad area)
and program-to-program switches 423
programming
guidelines 181
mixed language 252
secondary indexing 293
programming guidelines, general 483
programs
DL/I image capture 171
DL/I test 147
online 41
TM batch 40
programs, BMP 509
protected resources 116
protocol, locking 93
PSB (program specification block) 347
APSB (allocate program specification block) 121
CMPAT=YES 39
defining subset pointers 520
description 10
format 389, 509
generated program specification block (GPSB), format 249
GPSB (generated program specification block), format 249
PCB (program communication block) 249
PCB, types of 509
sample 678
scheduling in a call-level program 62
pseudo-abend 53
PSINDEX (Partitioned Secondary Index) 83
publications
changes xiii
organization xiii
titles xiii
PURG system service call 448

Q

Q command code 291
QC status code 44
qualification statement
coding 240
field name 182
field value 182
SSA qualification statement 182
multiple qualification statements 186
DEDB 188
HDAM 188
PHDAM 188
qualification statement
field value 182
randomizing routine
exit routine 188
relational operator 182
segment name 182
structure 182
qualified calls
overview 182
qualified SSA
structure with command code 189
qualified SSAs (segment search arguments)
qualification statement 182
qualifying
DL/I calls with command codes 189
SSAs 182
quantitative relationship between data aggregates 28

R

read access, specify with PROCOPT operand 93
read without integrity 95
read-only access, specify with PROCOPT operand 93
reading segments in MSDBs 321, 323
receiving messages
other IMS TM systems 414
record
database processing 12
database, description of 5
record descriptor word (RDW)
IMS Spool API 449
recording
data availability 20
information about your program 177
recoverable in-doubt structure. 278
recoverable resources 116
recovering databases 513
recovery
considerations in conversations 105
identifying requirements 50
in a batch-oriented BMP 42
in batch programs 39
RIS 278
token
definition 278
recovery EXEC DLI commands
XRST 514
recovery of databases 67
Recovery process
distributed 116
Recovery process *(continued)*
local 116
Recovery, Resource 116
redundant data 3
reestablish position in database 50
related (terminal related) MSDBs 321
relational database
hierarchical database, compared to 543
relational databases 36
relational operators
Boolean operators 186
list 182
overview 182
SSA qualification statement 182
SSA, coding 240
relationships
between data elements 22
data, hierarchical 5
defining logical 86
mapping data 28
relationships between data aggregates 28
remote DL/I 58
repetitive data elements, isolating 23
REPL (Replace) call
with MSDB, DEDB or VSO DEDB 323
reply to the terminal in a conversation 104
replying to the terminal in a conversation 422
report of instructor schedules 29
reports, creating 20
requests, processing 12
required application data, analyzing 17
requirements, analyzing processing 35
reserving
place for command codes 337
segment
command code 291
lock management 291
residency mode (RMODE) 252, 391
resolving data structure conflicts 81
resource managers 116
Resource Recovery
application program 116
Introduction to 116
protected resources 116
recoverable resources 116
resource managers 116
sync-point manager 116
Resource Recovery Services/Multiple Virtual Storage (RRS)
introduction to 116
Resource Recovery Services. 276
resources
protected 116
recoverable 116
security 15
response mode, description 107
restart your program
code for, description 68
with basic CHKP 50
with symbolic CHKP 50
Restart, Extended 49, 68

restarting your program, basic
checkpoints 284
restrictions
CHKP and XRST with GSAM 312
database calls
to DEDBs 337
to MSDBs 321
XRST (Extended Restart) call with
GSAM 312
ResultSet
iterating 698
ResultSet.getAsciiStream method 686
ResultSet.getCursorName method 686
ResultSet.getUnicodeStream method 686
retrieval of IMS database statistics 151
retrieving
dependent segments 86
RETRY option 55
return codes
UIB 237
RIS (recoverable in-doubt structure) 278
risks to security, combined files 3
RMODE 252
ROLB
in MPPs and transaction-oriented
BMPs 284
ROLB (Roll Back) call
compared to ROLL call 285
description 284
maintaining database integrity 284
usage 284
ROLB call 431
description 432
ROLB command
compared to ROLL and ROLS 431
ROLB system service call 39, 67
ROLB, ROLL, ROLS 431
ROLL (Roll) call
compared to ROLB call 285
description 284
maintaining database integrity 284
roll back point 431
ROLL call 431
description 432
ROLL command
compared to ROLB and ROLS 431
ROLL system service call 67
ROLS
backing out to an intermediate
backout point 288
ROLS (Roll Back to SETS) call
maintaining database integrity 284
TOKEN 284, 287
ROLS (Rollback to SETS or SETU)
command
backout point, intermediate 514
ROLS call 431
with LU 6.2 434
with TOKEN 434
without TOKEN 434
ROLS command
compared to ROLB and ROLL 431
ROLS system service call 39, 53, 70
root anchor point 75
root segment, definition 5
roster, current 17
routines
ESTAE 55
STAE 55
rows
relational representation, in 543
segment instances, compared to 543
RRS (z/OS Resource Recovery
Services) 276, 347
summary of IMS support 119
RSA (record search argument)
GSAM, reference 245
rules
coding an SSA 240

S

SAA resource recovery interface
calls 275
SAMETRM=YES 422
sample programs
call-level assembler language
CICS online 194
call-level COBOL, CICS online 201
call-level PL/I, CICS online 210
sample programs, command level
assembler language 484
C 495
COBOL 488
PL/I 491
schedule a PSB, in a call-level program,
how to 62
schedule, classes example 29
screen design considerations 102
SCS1 devices
meaning of designation 470
SCS2 devices
meaning of designation 470
SDSF (Spool Display and Search
Facility) 449
secondary indexes
multiple qualification statements 294
secondary indexing
DB PCB contents 296
description 82
effect on programming 293
examples of uses 83
information returned by DL/I 296
Partitioned Secondary Index
(PSINDEX) 83
specifying 83
SSAs 293
status codes 296
secondary logical unit 470
secondary processing sequence 293
security 99
and the PROCOPT= operand 93
database 91
field level sensitivity 91
identifying online requirements 99
key sensitivity 91
of databases and data
communications 15
of resources 15
password security 99
risks of combined files 3
segment sensitivity 91
signon 99
security *(continued)*
supplying information about your
application 99
terminal 99
security checks in program-to-program
switching 411
segment
description 5
preventing access to by other
programs 64
sensitivity 91
sequential dependent
identifying free space 529
locating a specific dependent 528
locating the last inserted
dependent 529
segment name
SSA qualification statement 182
segment search argument (SSA)
coding rules 240
segment search arguments (SSAs) 182
segment, information needed 191
segments
in medical database example 5
in SQL queries 543
medical database example 5
tables, compared to 543
Segments
Message Input Format 394
SELECT keyword
example query 543
selective partition processing
DFSHALDB ddname 301
HALDB control statement 301
sending messages
defining alternate PCBs for 408
other IMS TM systems 415
overview 393
to other application programs 411
to other IMS TM systems 413
using ISRT 408
sensitivity
data 10
field level 10, 91
general description 91
key 91
program 53
segment 91
sequence field
virtual logical child, in 182
sequential access methods
characteristics of 78
HISAM 79
HSAM 79
types of 78
sequential dependent segments
how stored 322
identifying free space 529
locating a specific dependent 528
locating the last inserted
dependent 529
POS (Position) command 528
sequential dependents 36, 322
overview 322
sequential processing only 79
SETO call
usage 452

SETO system service call 447
SETS
  backing out to an intermediate backout point 288
SETS call
  description 435
SETS system service call 39, 53, 70
SETU
  backing out to an intermediate backout point 288
SETU system service call 70
shared queues option 99
SHISAM (Simple Hierarchical Indexed Sequential Access Method) 80
SHSAM (Simple Hierarchical Sequential Access Method) 80
signon security 99
simple HISAM (SHISAM) 80
simple HSAM (SHSAM) 80
single mode 41, 46, 50
skeleton programs
  assembler language 192, 439
  C language 196, 439
  COBOL 199, 441
  Pascal 205, 442
  PL/I 207, 444
skills report, instructor 29
SLU 470
  type 1
    defining to operate with MFS 470
  type 2
    defining to operate with MFS 470
  type 6.1
    defining to operate with MFS 470
  type P
    defining to operate with MFS 470
SPA (scratchpad area) 105
specification of
  field level sensitivity 81
  frequency, checkpoint 52
specifying
  DB PCB mask 230
  GSAM data set attributes 316
  processing options for DEDBs 336
SPIE routine 55
Spool API
  CHNG call, keywords 452
  code examples
    Application PCB structure 457
    CHNG call to alternate PCB 457
    GU call to I/O PCB 457
    ISRT call to alternate PCB 457
  error codes
    description 452
    diagnosis, examples 454
  parsing errors
    diagnosis, examples 454
    error codes 452
    status codes 452
  print data set characteristics 452
  SETO call, keywords 452
  status codes 452
Spool Display and Search Facility (SDSF) 449
SQL (Structured Query Language) 36
  DELETE 691
  example query 543
SQL (Structured Query Language) *(continued)*
  INSERT 691
  prepared statements 696
  UPDATE 692
SQLException 700
SQLstate 700
SSA (segment search argument)
  coding
    formats 241
    restrictions 240
    rules 240
  coding rules 240
  command codes 189
  qualification statement 240
  reference 240
  relational operators 182
  restrictions 240
  segment name field 240
  structure with command code 189
  usage
    command codes 189
    guidelines 185
    multiple qualification statements 186
  virtual logical child 182
SSAs (segment search argument)
  overview 182
  segment name field 182
SSAs (segment search arguments) 182
  definition 182
  unqualified 182
  usage
    secondary indexing 293
STAE routines 55
standard application programs and MSC 428
STAT call
  formats for statistics
    OSAM buffer pool, STAT call 152
    OSAM buffer subpool, enhanced STAT call 156
    VSAM buffer subpool, enhanced STAT call 161
    VSAM buffer subpool, STAT call 154
  system service 355
  use in debugging 151, 174
Statement object
  retrieving 698
statistics, database 151
status code, QC 44
status codes
  AJ 527
  AM 527
  FSA 324
  GSAM 311
  H processing option 337
  logical relationships 299
  P processing option 336, 530
  subset pointers 332, 527
STATUS statement 147
storage of data
  in a combined file 3
  in a database 3
  in separate files 3
storage overlap 507
structure
  data 10
  physical, of a database 10
structure of data, methods 22
Structured Query Language (SQL) 36
subset pointer command codes
  restrictions 189
subset pointers
  command codes
    subset pointers 328
  DEDB
    managed by command codes 189
  defining DBD 520
  defining PSB 520
  defining, DBD 328
  defining, PCB 328
  description 328, 517
  MOVENEXT option 520
  moving forward 520
  preparation for using 519
  preparing to use 328
  specifying
    command codes for DEDBs 328
  status codes 332, 527
  using 328
summary of command codes 189
summary of symbolic CHKP and basic CHKP 49
supply security information, how to 99
symbolic checkpoint
  description 49, 65
  IDs, specifying 65
  issuing 68
  restart 68
  restart with 50
Symbolic Checkpoint (SYMCHKP) command
  restart 514
  XRST 514
SYMCHKP (Symbolic Checkpoint) command
  restart 514
  XRST 514
sync point
  application program 275
  CPI Communications driven programs 275
  data propagation 280
  log records 279
  relationship to commit point and check point 275
SYNC_LEVEL 275
sync_level values 115
sync-point manager (SPM) 116
synchronization point 431
synchronization point manager 115
synchronous conversation, description for LU 6.2 transactions 113
synonym, data element 19
syntax diagram
  how to read x
sysplex data-sharing 42
system log
  on tape 39
  storage 39
system service calls 347
  CHNG 447

system service calls *(continued)*
INIT 53
INQY 53
ISRT 447
LOG 165, 355
PURG 448
ROLB 39, 67
ROLB call 432
ROLL 67
ROLL call 432
ROLS 39, 53, 70
SETO 447
SETS 39, 53, 70
SETU 70
STAT 151, 355

T

tables
relational representation, in 543
segments, compared to 543
take checkpoints, how to 65
terminal screen, designing 102
terminal security 99
termination of a PSB, restrictions 62
termination, abnormal 46
test of application programs
using BTS 148
using DFSDDLT0 171
using DL/I test program 147
what you need 147, 169
test of DL/I call sequences 147, 171
test, unit 147
testing
CICS programs
tools 169
timeout
activating 413
TM batch program 40
token, definition of 105
tracing
methods of the Java class libraries for IMS 702
Trace statements, adding 702
XMLTrace 700
trademarks 707
TRANSACT macro 46
transaction code 41
transaction response mode 41
transaction-oriented BMPs
ROLB 284
transaction-oriented BMPs. 50
translator
options required for EXEC DLI 501
TREATMNT segment 5
TSO application programs 46
two-phase 276
two-phase commit
overview 276
single-phase 279
UOR 278
two-phase commit process
UOR 116
two-phase commit protocol 116
TXTU parameter 448
type 18 log record 65

U

UIB (user interface block)
defining, in program 237
field names 237
PCB address list, accessing 237
return codes, accessing 237
unavailability of data 53, 69
unique identifier, data 19
unit of recovery (UOR)
definition 278
unit of work 46
unit of work (UOW)
crossing a boundary when processing DEDBs 530
unit test 147
unqualified calls
command codes
C 182
SSAs (segment search arguments) 182
definition 182
DL/I calls (general information)
types 182
overview 182
qualified calls
definition 182
SSAs (segment search arguments)
qualified 182
unqualified 182
unqualified SSA
structure with command code 189
usage with command codes 189
unqualified SSAs
segment name field 182
UOR (unit of recovery) 116
definition 278
in-doubt
definition 278
in-flight
definition 278
UOW (unit of work)
crossing a boundary when processing DEDBs 530
UOW boundary, processing DEDB 336, 337
update access, specify with PROCOPT operand 93
UPDATE keyword 692
example 692
updating
segments in an MSDB, DEDB or VSO DEDB 323
uppercase, using Basic Edit 399
user interface block 237
user requirements, analyzing 15
utilities
Batch Backout 39
DFSERA10 65, 355
File Select and Formatting Print program 49

V

values, isolating duplicate 23
variable-length database segments
IMS Universal drivers 555
variable-length database segments *(continued)*
SQL support for 555
VBASF, formatted VSAM subpool statistics 154
VBASS, formatted summary of VSAM subpool statistics 154
VBASU, unformatted VSAM subpool statistics 154
VBESF, formatted VSAM subpool statistics 161
VBESS, formatted summary of VSAM subpool statistics 161
VBESU, unformatted VSAM subpool statistics 161
view of data, a program's 10
view, local 29
virtual logical child 182
VisualGen 20
VSAM buffer subpool, retrieving
enhanced subpool statistics 161
statistics 154, 161
VTAM I/O facility
effects on VTAM terminals 413
VTAM terminal
activating a timeout 413

W

wait-for-input (WFI)
transactions 41, 44
waits, program 50
WebSphere Application Server
adding permissions to server.policy file 700
WebSphere Application Server for z/OS applications
bean-managed EJBs 670
container-managed EJBs 672
Java servlets 673
programming models 670, 672, 673
WFI parameter 44
writing information to the system log 165

X

X'18' log record 49
XML tracing
security requirements 700
XMLTrace
application 702
enabling 701
XMLTrace class 700
XMLTrace.enable 701
XMLTrace.libTraceLevel 702
XMLTrace.methods of the Java class libraries for IMS 702
XRST (Extended Restart) 49

Z

z/OS
extended addressing capabilities
addressing mode (AMODE) 391
DCCTL environment 391

z/OS *(continued)*
extended addressing capabilities
(continued)
preloaded program 391
residency mode (RMODE) 391
z/OS files
access to 36, 58
description 58
z/OS Resource Recovery Services 121, 275
z/OS Resource Recovery Services (RRS)
ODBA interface 347
summary of IMS support 119
z/OS Scheduler JCL Facility (SJF) 448
Z1 field 395
Z2 field 395
ZZ field
in input message 394
in output message 395

Product Number: 5635-A02

Printed in USA

SC19-2428-05

Spine information:

IMS **Version 11** **Application Programming**