

IMS



# Application Programming: Database Manager

*Version 9*



IMS



# Application Programming: Database Manager

*Version 9*

**Note**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 383

**Quality Partnership Program (QPP) Edition (December 2003) (Softcopy Only)**

This QPP edition applies to Version 9 of IMS (product number 5655-J38) and to all subsequent releases and modifications until otherwise indicated in new editions.

**© Copyright International Business Machines Corporation 1974, 2003. All rights reserved.**

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

# Contents

<b>Figures</b> . . . . .	vii
<b>Tables</b> . . . . .	ix
<b>About This Book</b> . . . . .	xi
Summary of Contents . . . . .	xi
Prerequisite Knowledge . . . . .	xi
How to Use This Book . . . . .	xii
Terminology . . . . .	xii
How to Read Syntax Diagrams . . . . .	xii
How to Send Your Comments . . . . .	xv
Change Indicators . . . . .	xv
<b>Summary of Changes</b> . . . . .	xvii
Changes to the Current Edition of This Book for IMS Version 9 . . . . .	xvii
Changes to This Book for IMS Version 9 . . . . .	xvii
Library Changes for IMS Version 9 . . . . .	xvii

---

## Part 1. Writing Application Programs. . . . . 1

<b>Chapter 1. How Application Programs Work with the IMS Database Manager</b> . . . . .	9
Application Program Environments . . . . .	9
The Application Programming Interface . . . . .	9
Getting Started with DL/I . . . . .	12
Getting Started with DL/I (for CICS Online Users) . . . . .	13
Getting Started with DL/I using the ODBA Interface . . . . .	15
DL/I Calls . . . . .	16
Sample Hierarchies . . . . .	19
SSA Overview . . . . .	24
Command Codes . . . . .	28
IVP Sample Application Program . . . . .	45
<b>Chapter 2. Writing Your Application Programs</b> . . . . .	47
Programming Guidelines . . . . .	47
Coding DL/I Calls and Data Areas . . . . .	48
Preparing to Run Your CICS DL/I Call Program . . . . .	49
Sample Programs . . . . .	49
<b>Chapter 3. Defining Application Program Elements</b> . . . . .	77
Formatting DL/I Calls for Language Interfaces . . . . .	77
Application Programming for Assembler Language . . . . .	78
Application Programming for C Language . . . . .	80
Application Programming for COBOL . . . . .	83
Application Programming for Pascal . . . . .	86
Application Programming for PL/I . . . . .	88
Relationship of Calls to PCBs . . . . .	91
Specifying the I/O PCB Mask . . . . .	92
Specifying the DB PCB Mask . . . . .	95
Specifying the AIB Mask . . . . .	98
Specifying the AIB Mask for ODBA Applications . . . . .	99
Specifying the UIB (CICS Online Programs Only) . . . . .	102
Specifying the I/O Areas . . . . .	105
Segment Search Arguments . . . . .	106

GSAM Data Areas . . . . .	111
The AIBTDLI Interface . . . . .	111
Specifying the Language Specific Entry Point . . . . .	112
PCB Lists . . . . .	115
The AERTLDI interface . . . . .	116
Language Environment . . . . .	117
Special DL/I Situations. . . . .	118
<b>Chapter 4. Writing DL/I Calls for Database Management . . . . .</b>	<b>121</b>
CIMS Call . . . . .	121
CLSE Call . . . . .	123
DEQ Call . . . . .	123
DLET Call . . . . .	125
FLD Call . . . . .	126
GN/GHN Call . . . . .	128
GNP/GHNP Call . . . . .	132
GU/GHU Call . . . . .	135
ISRT Call . . . . .	138
OPEN Call . . . . .	141
POS Call . . . . .	142
REPL Call . . . . .	145
<b>Chapter 5. Writing DL/I Calls for System Services . . . . .</b>	<b>149</b>
APSB Call . . . . .	150
CHKP (Basic) Call . . . . .	150
CHKP (Symbolic) Call . . . . .	151
DPSB Call . . . . .	153
GMSG Call . . . . .	154
GSCD Call . . . . .	156
ICMD Call . . . . .	157
INIT Call . . . . .	159
INQY Call . . . . .	163
LOG Call . . . . .	169
PCB Call (CICS Online Programs Only) . . . . .	171
RCMD Call . . . . .	172
ROLB Call . . . . .	173
ROLL Call . . . . .	174
ROLS Call . . . . .	175
SETS/SETU Call. . . . .	176
SNAP Call . . . . .	177
STAT Call . . . . .	180
SYNC Call . . . . .	182
TERM Call (CICS Online Programs Only) . . . . .	183
XRST Call . . . . .	184
<b>Chapter 6. Monitoring Your Position in the Database . . . . .</b>	<b>189</b>
Understanding Current Position in the Database . . . . .	189
Current Position after Unsuccessful Calls. . . . .	194
<b>Chapter 7. Multiple Qualification Statements . . . . .</b>	<b>199</b>
Overview of Multiple Qualification Statements . . . . .	199
Example using Multiple Qualification Statements . . . . .	200
Multiple Qualification Statements for HDAM, PHDAM, or DEDB . . . . .	201
<b>Chapter 8. Multiple Processing . . . . .</b>	<b>203</b>
Multiple Positioning . . . . .	203

Advantages of Using Multiple Positioning . . . . .	206
Using Multiple DB PCBs . . . . .	208
<b>Chapter 9. Secondary Indexing and Logical Relationships . . . . .</b>	<b>211</b>
How Secondary Indexing Affects Your Program . . . . .	211
Processing Segments in Logical Relationships . . . . .	214
<b>Chapter 10. Processing GSAM Databases . . . . .</b>	<b>219</b>
Accessing GSAM Databases . . . . .	219
GSAM Record Formats . . . . .	222
GSAM I/O Areas . . . . .	223
GSAM Status Codes . . . . .	223
Symbolic CHKP and XRST with GSAM . . . . .	224
GSAM Coding Considerations . . . . .	224
Origin of GSAM Data Set Characteristics . . . . .	225
<b>Chapter 11. Processing Fast Path Databases . . . . .</b>	<b>229</b>
Fast Path Database Calls . . . . .	229
MSDBs and DEDBs: Overview . . . . .	230
Processing MSDBs and DEDBs . . . . .	231
Restrictions on Using Calls for MSDBs . . . . .	237
Processing DEDBs (IMS, CICS with DBCTL) . . . . .	238
Restrictions on Using Calls for DEDBs. . . . .	246
Fast Path Coding Considerations. . . . .	247
<b>Chapter 12. Recovering Databases and Maintaining Database Integrity . . . . .</b>	<b>249</b>
Issuing Checkpoints . . . . .	249
Restarting Your Program and Checking for Position . . . . .	249
Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)	250
Reserving Segments for the Exclusive Use of Your Program. . . . .	256

---

**Part 2. IMS Adapter for REXX . . . . . 259**

<b>Chapter 13. IMS Adapter for REXX . . . . .</b>	<b>261</b>
Addressing Other Environments . . . . .	262
REXX Transaction Programs . . . . .	262
REXXTDLI Commands . . . . .	266
REXXTDLI Calls . . . . .	267
REXXIMS Extended Commands . . . . .	270
<b>Chapter 14. Sample Execs Using REXXTDLI . . . . .</b>	<b>283</b>
SAY Exec: For Expression Evaluation . . . . .	283
PCBINFO Exec: Display PCBs Available in Current PSB . . . . .	284
PART Execs: Database Access Example . . . . .	286
DOCMD: IMS Commands Front End . . . . .	289
IVPREXX: MPP/IFP Front End for General Exec Execution . . . . .	293

---

**Part 3. Reference . . . . . 295**

<b>Chapter 15. Summary of DM and System Service Calls . . . . .</b>	<b>297</b>
Database Management Call Summary . . . . .	297
System Service Call Summary. . . . .	298
<b>Chapter 16. Command Codes Reference . . . . .</b>	<b>301</b>

<b>Chapter 17. CICS-DL/I User Interface Block Return Codes</b> . . . . .	303
Not-Open Conditions . . . . .	304
Invalid Request Conditions . . . . .	304

---

<b>Part 4. Appendixes</b> . . . . .	<b>305</b>
-------------------------------------	------------

<b>Appendix A. Sample Exit Routine (DFSREXXU)</b> . . . . .	307
---	-----

<b>Appendix B. The DL/I Test Program (DFSDDLTO)</b> . . . . .	309
---	-----

Control Statements . . . . .	309
Planning the Control Statement Order . . . . .	311
ABEND Statement . . . . .	311
CALL Statement . . . . .	312
COMMENT Statement. . . . .	332
COMPARE Statement . . . . .	333
IGNORE Statement. . . . .	339
OPTION Statement . . . . .	339
PUNCH Statement . . . . .	340
STATUS Statement . . . . .	342
WTO Statement . . . . .	345
WTOR Statement . . . . .	346
JCL Requirements . . . . .	346
Execution of DFSDDLTO in IMS Regions . . . . .	350
Explanation of DFSDDLTO Return Codes . . . . .	351
Hints on Using DFSDDLTO . . . . .	351

<b>Appendix C. The Database Resource Adapter (DRA)</b> . . . . .	355
--	-----

Thread Concepts . . . . .	355
Sync Points . . . . .	358
The DRA Startup Table . . . . .	362
Enabling the DRA for a CCTL . . . . .	363
Enabling the DRA for the ODBA Interface . . . . .	364
Processing CCTL DRA Requests . . . . .	365
Processing ODBA Calls . . . . .	366
CCTL-Initiated DRA Function Requests . . . . .	366
PAPL Mapping Format . . . . .	375
Terminating the DRA . . . . .	375
Designing the CCTL Recovery Process . . . . .	375
CCTL Performance—Monitoring DRA Thread TCBS . . . . .	376

<b>Notices</b> . . . . .	383
--------------------------	-----

Programming Interface Information . . . . .	385
Trademarks. . . . .	385
Product Names . . . . .	386

<b>Bibliography</b> . . . . .	387
-------------------------------	-----

IMS Version 9 Library . . . . .	387
---------------------------------	-----

<b>Index</b> . . . . .	389
------------------------	-----



# Figures

1. Hierarchical Relationship of Application Programming Books . . . . .	xii
2. Application View of DB/DC Environment . . . . .	10
3. Application View of DBCTL Environment . . . . .	11
4. DL/I Program Elements . . . . .	12
5. The Structure of a Call-Level CICS Online Program . . . . .	14
6. Normal Relationship between Programs, PSBs, PCBs, DBDs, and Databases . . . . .	19
7. Relationship between Programs and Multiple PCBs (Concurrent Processing). . . . .	19
8. Medical Hierarchy . . . . .	20
9. Segment with a Noncontiguous Sequence Field . . . . .	26
10. D Command Code Example. . . . .	27
11. U Command Code Example. . . . .	37
12. Processing for the Passbook Example . . . . .	40
13. Moving the Subset Pointer to the Next Segment after Your Current Position . . . . .	41
14. Retrieving the First Segment in a Chain of Segments . . . . .	42
15. Unconditionally Setting the Subset Pointer to Your Current Position . . . . .	43
16. Conditionally Setting the Subset Pointer to Your Current Position . . . . .	44
17. Sample Assembler Language Program. . . . .	51
18. Sample Call-Level Assembler Language Program (CICS Online) . . . . .	54
19. Sample C Language Program . . . . .	57
20. Sample COBOL Program. . . . .	61
21. Sample Call-Level OS/V COBOL program (CICS Online) . . . . .	66
22. Sample Pascal Program . . . . .	69
23. Sample PL/I Program . . . . .	71
24. Sample Call-Level PL/I Program (CICS Online). . . . .	75
25. Defining the UIB, PCB Address List, and the PCB Mask for VS COBOL II . . . . .	103
26. Defining the UIB, PCB Address List, and the PCB Mask for OS/VS COBOL . . . . .	104
27. The COBOL DLIUIB Copy Book. . . . .	104
28. Defining the UIB, PCB Address List, and the PCB Mask for PL/I . . . . .	105
29. Defining the UIB, PCB Address List, and the PCB Mask for Assembler Language . . . . .	105
30. Example Code: * CONSTANT AREA . . . . .	108
31. Qualified SSA without Command Codes . . . . .	110
32. Hierarchic Sequence . . . . .	130
33. I/O Area for SNAP Operation Parameters . . . . .	178
34. Current Position Hierarchy . . . . .	190
35. Example Code: Deleting Segment C11 . . . . .	191
36. Hierarchy after Deleting a Segment . . . . .	192
37. Hierarchy after Deleting a Segment and Dependents . . . . .	192
38. Hierarchy after Adding New Segments and Dependents . . . . .	194
39. DL/I Positions . . . . .	195
40. Multiple Processing . . . . .	203
41. Multiple Positioning Hierarchy. . . . .	204
42. Single and Multiple Positioning Hierarchy . . . . .	205
43. Example of Using the Dependent AND . . . . .	213
44. Example of Using the Independent AND. . . . .	213
45. Patient and Item Hierarchies . . . . .	215
46. Concatenated Segment . . . . .	216
47. //IMS DD Statement Example. . . . .	228
48. Sample PCB Specifying View=MSDB . . . . .	236
49. Processing a Long Chain of Segment Occurrences with Subset Pointers. . . . .	238
50. Examples of Setting Subset Pointers . . . . .	239
51. Additional Examples of Setting Subset Pointers . . . . .	239
52. How Subset Pointers Divide a Chain into Subsets . . . . .	240
53. SETS and ROLS Calls Working Together . . . . .	254

54.	JCL Code Used to Run the IVPREXX Sample Exec . . . . .	264
55.	IMS Adapter for REXX Logical Overview Diagram . . . . .	265
56.	Exec To Do Calculations . . . . .	283
57.	PDF EDIT Session on the SAY Exec . . . . .	284
58.	Example Output from the SAY Exec . . . . .	284
59.	Example Output of PCBINFO Exec on a PSB without Database PCBs. . . . .	284
60.	Example Output of PCBINFO Exec on a PSB with a Database PCB. . . . .	284
61.	PCBINFO Exec Listing . . . . .	285
62.	Example Output of PARTNUM Exec . . . . .	286
63.	Example Output of PARTNAME Exec . . . . .	286
64.	PARTNUM Exec: Show Set of Parts Near a Specified Number . . . . .	287
65.	PARTNAME Exec: Show Parts with Similar Names. . . . .	288
66.	Output from = > DOCMD . . . . .	289
67.	Output from = > DOCMD /DIS NODE ALL;? . . . . .	289
68.	Output from = > DOCMD /DIS NODE ALL;CID>0 . . . . .	289
69.	Output from = > DOCMD /DIS NODE ALL;TYPE=SLU2 . . . . .	290
70.	Output from = > DOCMD /DIS TRAN ALL;ENQCT>0 & RECTYPE='T02'. . . . .	290
71.	Output from = > DOCMD /DIS LTERM ALL;ENQCT>0 . . . . .	290
72.	DOCMD Exec: Process an IMS Command . . . . .	291
73.	Example JCL Code for DD Statement Definition . . . . .	347
74.	Example JCL Code for DFSDDLTO in a BMP . . . . .	347
75.	ODBA Two-Phase Sync Point Processing . . . . .	361
76.	DRA Component Structure with the ODBA Interface . . . . .	365

## Tables

1. How to Read Syntax Diagrams . . . . .	xiii
2. PATIENT Segment . . . . .	20
3. ILLNESS Segment . . . . .	21
4. TREATMNT Segment . . . . .	21
5. BILLING Segment . . . . .	21
6. PAYMENT Segment. . . . .	21
7. HOUSEHOLD Segment . . . . .	22
8. Teller Segment in a Fixed Related MSDB . . . . .	22
9. Branch Summary Segment in a Dynamic Related MSDB . . . . .	23
10. Account Segment in a Nonrelated MSDB . . . . .	23
11. Qualified SSA Structure . . . . .	24
12. Unqualified SSA with Command Code . . . . .	27
13. Qualified SSA with Command Code . . . . .	28
14. Command Codes for DL/I Calls . . . . .	28
15. Call Relationship to PCBs . . . . .	91
16. I/O PCB Mask . . . . .	92
17. DB PCB Mask . . . . .	95
18. AIB Fields . . . . .	98
19. AIB Fields for ODBA Applications' Use . . . . .	99
20. Relational Operators . . . . .	107
21. I/O PCB and Alternate PCB Information Summary . . . . .	116
22. Using LANG= Option in a Language Environment for PL/I Compatibility . . . . .	118
23. Unqualified POS Call: Keywords and Map of the I/O Area Returned . . . . .	143
24. GMSG Support by Application Region Type . . . . .	156
25. ICMDB Support by Application Region Type . . . . .	158
26. INIT DBQUERY: Examples for ASMTDLI, CBLTDLI, CTDLI, and PASTDLI . . . . .	160
27. INIT DBQUERY: I/O Area Example for PLITDLI . . . . .	160
28. INIT I/O Area Examples for ASMTDLI, CBLTDLI, CTDLI, and PASTDLI . . . . .	161
29. INIT I/O Area Examples for PLITDLI . . . . .	161
30. INIT I/O Area Examples for ASMTDLI, CBLTDLI, CTDLI, and PASTDLI . . . . .	162
31. INIT I/O Area Examples for PLITDLI . . . . .	162
32. INQY ENVIRON Data Output . . . . .	166
33. Subfunction, PCB, and I/O Area Combinations for the INQY Call. . . . .	169
34. Log Record Formats for COBOL, C, Assembler, Pascal, and PL/I Programs for the AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI Interfaces. . . . .	170
35. Log Record Formats for COBOL, C, Assembler, Pascal, and PL/I Programs for the PLITDLI Interface . . . . .	170
36. RCMD Support by Application Region Type . . . . .	173
37. SNAP Operation Parameters . . . . .	179
38. Results of Single and Multiple Positioning with DL/I Calls . . . . .	205
39. GSAM DB PCB Mask . . . . .	220
40. Summary of GSAM Calls . . . . .	225
41. Summary of Fast Path Database Calls . . . . .	229
42. Subset Pointer Command Codes and Calls . . . . .	230
43. FSA Structure . . . . .	232
44. Unqualified SSA with Subset Pointer Command Code. . . . .	241
45. Qualified SSA with Subset Pointer Command Code . . . . .	241
46. Qualified POS Call: Keywords and Map of I/O Area Returned . . . . .	243
47. Comparison of ROLB, ROLL, and ROLS . . . . .	251
48. IMS Adapter for REXX Parameter Types and Definitions . . . . .	268
49. REXXIMS Extended Commands. . . . .	270
50. Summary of DB Calls . . . . .	297
51. Summary of System Service Calls . . . . .	298

52. Summary of Command Codes . . . . .	301
53. Command Codes and Calls . . . . .	301
54. Return Codes in UIBFCTR . . . . .	303
55. Return Codes in UIBDLTR if UIBFCTR='0C' (NOTOPEN) . . . . .	303
56. Return Codes in UIBDLTR if UIBFCTR='08' (INVREQ) . . . . .	303
57. Summary of DFSDDLTO Control Statements . . . . .	309
58. ABEND Statement . . . . .	311
59. CALL FUNCTION Statement . . . . .	312
60. CALL DATA Statement . . . . .	315
61. OPTION DATA Statement . . . . .	317
62. FEEDBACK DATA Statement . . . . .	317
63. DL/I Call Functions . . . . .	318
64. CALL FUNCTION Statement (Column-Specific SSAs) . . . . .	329
65. CALL FUNCTION Statement with DFSDDLTO Call Functions . . . . .	331
66. COMMENT Statement . . . . .	332
67. COMPARE DATA Statement . . . . .	333
68. COMPARE AIB Statement . . . . .	335
69. COMPARE PCB Statement . . . . .	335
70. IGNORE Statement . . . . .	339
71. OPTION Statement . . . . .	339
72. PUNCH CTL Statement . . . . .	340
73. STATUS Statement . . . . .	342
74. WTO Statement . . . . .	345
75. WTOR Statement . . . . .	346
76. Example of Events in a Multithreading System . . . . .	357
77. CCTL Single-Phase Sync Point Processing . . . . .	360
78. CCTL Two-Phase Sync Point Processing . . . . .	360
79. Information Provided for the Schedule Process: . . . . .	377
80. Information Provided at UOR Termination: . . . . .	377

---

## About This Book

This softcopy book is available only in PDF and BookManager<sup>®</sup> formats. This book is available on the IMS Version 9 Licensed Product Kit (LK3T-7213). To get the most current versions of the PDF and BookManager formats, go to the IMS Web site at [www.ibm.com/ims](http://www.ibm.com/ims) and link to the Library page.

This book is a guide to application programming in an IMS<sup>™</sup> Database Manager (IMS DB) environment. It covers basic information on coding DL/I calls for DB programs. The book is designed to provide guidance for application programmers who use the IMS DB environment to create and run application programs. Portions of this book are for programmers who use IMS from a Customer Information Control System (CICS<sup>®</sup>) environment.

This book also contains information on the DBCTL environment. DBCTL is generated by IMS DB, contains no data communication components, and is designed to function as a database manager for non-IMS transaction management systems.

---

## Summary of Contents

This book has four parts:

- Part 1, "Writing Application Programs," on page 1 provides basic information on coding DL/I calls for IMS DB programs.
- Part 2, "IMS Adapter for REXX," on page 259 provides information that you can use to interactively develop REXX EXECs under TSO/E and execute them in IMS MPPs, BMPs, IFPs, or batch regions.
- Part 3, "Reference," on page 295 provides additional information you need to write your application program.
- Part 4, "Appendixes" contains appendixes on several subjects including sample exit routines, sample applications, and use of the DL/I test program (DFSDDLTO).

---

## Prerequisite Knowledge

IBM<sup>®</sup> offers a wide variety of classroom and self-study courses to help you learn IMS. For a complete list, see the IMS home page on the World Wide Web at: [www.ibm.com/ims](http://www.ibm.com/ims).

Before using this book, you should understand the concepts of application design presented in *IMS Version 9: Application Programming: Design Guide*, which assumes you understand basic IMS concepts and the various environments.

This book is an extension to *IMS Version 9: Application Programming: Design Guide*. The IMS concepts explained in this manual are limited to those concepts that are pertinent to developing and coding application programs. You should also know how to use assembler language, C language, COBOL, Pascal, or PL/I. CICS programs can be written in assembler language, C language, COBOL, PL/I, and C++.

---

## How to Use This Book

This book is one of several books documenting the IMS application programming task. The complete package of application programming materials is as follows:

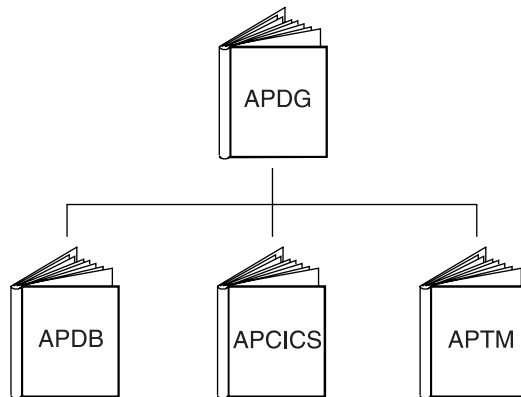


Figure 1. Hierarchical Relationship of Application Programming Books

- *IMS Version 9: Application Programming: Design Guide (APDG)*, is the introductory application programming book and is also the place to find information common to all of the application programming environments.
- *IMS Version 9: Application Programming: Database Manager (APDB)* describes how to write an application program to process a database using DL/I calls. This book applies to both IMS and CICS environments.
- *IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS (APCICS)* describes how to write an application program to process the database using EXEC DLI commands.
- *IMS Version 9: Application Programming: Transaction Manager (APT)* describes how to write an application program to process messages using DC calls.

For definitions of terms used in this manual and references to related information in other manuals, see the *IMS Version 9: Master Index and Glossary*.

---

## Terminology

In this manual, the term *external subsystems* refers to subsystems that are not CCTL subsystems, unless indicated otherwise. One example of an external subsystem is DB2®.

For definitions of terminology used in this manual and references to related information in other manuals, see *IMS Version 9: Master Index and Glossary*.

---

## How to Read Syntax Diagrams

Each syntax diagram in this book begins with a double right arrow and ends with a right and left arrow pair. Lines that begin with a single right arrow are continuation lines. You read a syntax diagram from left to right and from top to bottom, following the direction of the arrows.

Conventions used in syntax diagrams are described in Table 1:

Table 1. How to Read Syntax Diagrams


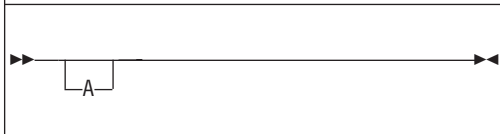
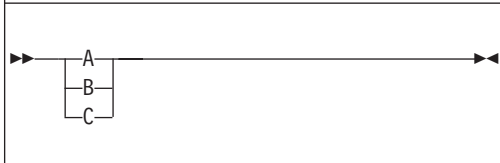
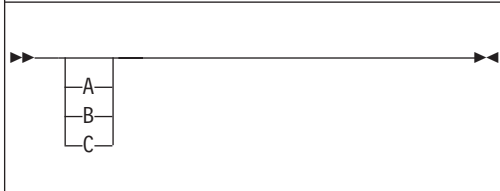

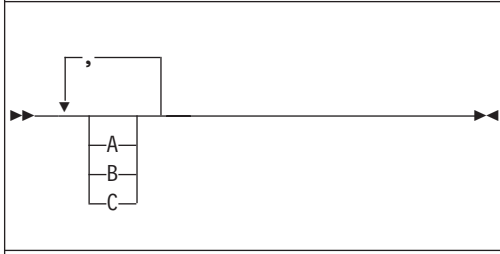
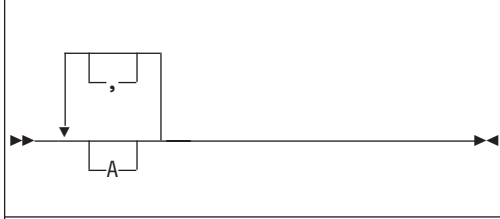
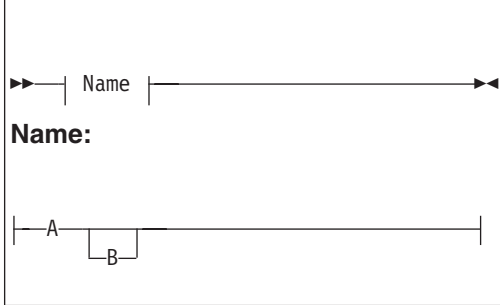
Convention	Meaning
	<p>You must specify values A, B, and C. Required values are shown on the main path of a syntax diagram.</p>
	<p>You have the option to specify value A. Optional values are shown below the main path of a syntax diagram.</p>
	<p>You must specify value A, B, or C.</p>
	<p>You have the option to specify A, B, C, or none of these values.</p>
	<p>You have the option to specify A, B, C, or none of these values. If you don't specify a value, A is the default.</p>
	<p>You have the option to specify one, more than one, or none of the values A, B, or C. Any required separator for multiple or repeated values (in this example, the comma) is shown on the arrow.</p>
	<p>You have the option to specify value A multiple times. The separator in this example is optional.</p>
	<p>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.</p>

Table 1. How to Read Syntax Diagrams (continued)

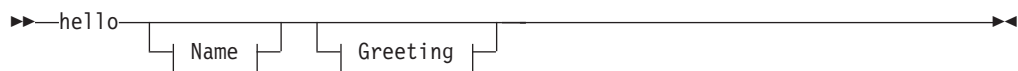
Convention	Meaning
Punctuation marks and numbers	Enter punctuation marks (slashes, commas, periods, parentheses, quotation marks, equal signs) and numbers exactly as shown.
Uppercase values	Keywords, their allowable synonyms, and reserved parameters, appear in uppercase letters for z/OS. Enter these values exactly as shown.
Lowercase values without italics	Keywords, their allowable synonyms, and reserved parameters, appear in lowercase letters for UNIX. Enter these values exactly as shown.
Lowercase values in italics (for example, <i>name</i> )	Supply your own text or value in place of the <i>name</i> variable.
b	A b symbol indicates one blank position.

Other conventions include the following:

- When entering commands, separate parameters and keywords by at least one blank if there is no intervening punctuation.
- Footnotes are shown by a number in parentheses, for example, (1).
- Parameters with number values end with the symbol #.
- Parameters that are names end with 'name'.
- Parameters that can be generic end with the symbol \*.

## Syntax Diagram Example

Here is an example syntax diagram that describes the **hello** command.



### Name:



### Greeting:



### Notes:

- 1 You can code up to three names.
- 2 Compose and add your own greeting (for example, how are you?).

According to the syntax diagram, these are all valid versions of the **hello** command:



```
hello
hello name
hello name, name
hello name, name, name
hello, your_greeting
hello name, your_greeting
hello name, name, your_greeting
hello name, name, name, your_greeting
```

The space before the *name* value is significant. If you do not code *name*, you must still code the comma before *your\_greeting*.

---

## 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 book or any other IMS documentation, you can do one of the following:

- Go to the IMS home page at: [www.ibm.com/ims](http://www.ibm.com/ims). There you will find an online feedback page where you can enter and submit comments.
- Send your comments by e-mail to [imspubs@us.ibm.com](mailto:imspubs@us.ibm.com). Be sure to include the name of the book, the part number of the book, the version of IMS, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

---

## Change Indicators

Symbols or numbers that appear to the left of the text are *change indicators*, which identify technical information that has changed between publication releases. The change indicators in this book identify changes as follows:

- I Technical changes are indicated in this publication by a vertical bar (I) to the left of the changed text.



---

## Summary of Changes

---

### Changes to the Current Edition of This Book for IMS Version 9

---

#### Changes to This Book for IMS Version 9

This edition is a draft version of this book intended for use during the Quality Partnership Program (QPP). Contents of this book are preliminary and under development.

This book contains new technical information for IMS Version 9, as well as editorial changes. This book contains new information about:

- Coding a batch program in COBOL
- Issuing a DL/I POS call

---

#### Library Changes for IMS Version 9

Changes to the IMS Library for IMS Version 9 include the addition of new titles, the change of one title, and a major terminology change.

#### New and Revised Titles

The following list details the major changes to the IMS Version 9 library:

- *IMS Version 9: HALDB Online Reorganization Guide and Reference*  
The library includes a new book: *IMS Version 9: HALDB Online Reorganization Guide and Reference*. This information is available only in PDF and BookManager formats.
- *IMS Version 9: An Introduction to IMS*  
The library includes a new book: *IMS Version 9: An Introduction to IMS*.
- The book formerly titled *IMS Version 8: IMS Java User's Guide* is now titled *IMS Version 9: IMS Java Guide and Reference*.

#### Terminology Changes

IMS Version 9 introduces new terminology for IMS commands:

##### **type-1 command**

A command, generally preceded by a leading slash character, that can be entered from any valid IMS command source. In IMS Version 8, these commands were called *classic* commands.

##### **type-2 command**

A command that is entered only through the OM API. Type-2 commands are more flexible and can have a broader scope than type-1 commands. In IMS Version 8, these commands were called *IMSplex* commands or *enhanced* commands.

#### Accessibility Enhancements

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products. The major accessibility features in z/OS products, including IMS, enable users to:

- Use assistive technologies such as screen readers and screen magnifier software

- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

### **User Assistive Technologies**

Assistive technology products, such as screen readers, function with the IMS user interfaces. Consult the documentation of the assistive technology products for specific information when you use assistive technology to access these interfaces.

### **Accessible Documentation**

Online information for IMS Version 9 is available in BookManager format, which is an accessible format. All BookManager functions can be accessed by using a keyboard or keyboard shortcut keys. BookManager also allows you to use screen readers and other assistive technologies. The BookManager READ/MVS product is included with the z/OS base product, and the BookManager Softcopy Reader (for workstations) is available on the IMS Licensed Product Kit (CD), which you can download from the Web at [www.ibm.com](http://www.ibm.com).

### **Keyboard Navigation of the User Interface**

Users can access IMS user interfaces using TSO/E or ISPF. Refer to the *z/OS V1R1.0 TSO/E Primer*, the *z/OS V1R1.0 TSO/E User's Guide*, and the *z/OS V1R1.0 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.

# Part 1. Writing Application Programs

<b>Chapter 1. How Application Programs Work with the IMS Database Manager</b>	<b>9</b>
Application Program Environments . . . . .	9
The Application Programming Interface . . . . .	9
The DB/DC Environment . . . . .	10
The DBCTL Environment . . . . .	10
The DB Batch Environment . . . . .	12
Getting Started with DL/I . . . . .	12
Getting Started with DL/I (for CICS Online Users) . . . . .	13
Getting Started with DL/I using the ODBA Interface . . . . .	15
Common Logic Flow for SRMS and MRMS . . . . .	15
Logic Flow for SRMS . . . . .	15
Logic Flow for MRMS . . . . .	16
DL/I Calls . . . . .	16
DB Call Functions . . . . .	16
System Service Call Functions . . . . .	16
Status Codes, Return Codes, and Reason Codes . . . . .	17
Exceptional Conditions . . . . .	18
High Availability Large Databases . . . . .	18
Error Routines . . . . .	18
DL/I and Your Application Program . . . . .	18
DBDs and PSBs . . . . .	18
SSAs and Command Codes . . . . .	19
Sample Hierarchies . . . . .	19
Medical Database Example . . . . .	20
Bank Account Example . . . . .	22
SSA Overview . . . . .	24
Unqualified SSAs . . . . .	24
Qualified SSAs . . . . .	24
Guidelines for Using SSAs . . . . .	26
SSAs and Command Codes . . . . .	27
Command Codes . . . . .	28
General Command Codes for DL/I Calls . . . . .	29
DEDB Command Codes for DL/I . . . . .	39
IVP Sample Application Program . . . . .	45
<b>Chapter 2. Writing Your Application Programs</b>	<b>47</b>
Programming Guidelines . . . . .	47
Coding DL/I Calls and Data Areas . . . . .	48
Program Design Considerations . . . . .	48
Checkpoint Considerations . . . . .	49
Segment Considerations . . . . .	49
Data Structure Considerations . . . . .	49
Preparing to Run Your CICS DL/I Call Program . . . . .	49
Sample Programs . . . . .	49
Coding a Batch Program in Assembler Language . . . . .	50
Coding a CICS Online Program in Assembler Language . . . . .	53
Coding a Batch Program in C Language . . . . .	56
Coding a Batch Program in COBOL . . . . .	60
Coding a CICS Online Program in COBOL . . . . .	63
Coding a Batch Program in Pascal . . . . .	68
Coding a Batch Program in PL/I . . . . .	71
Coding a CICS Online Program in PL/I . . . . .	74

<b>Chapter 3. Defining Application Program Elements</b>	77
Formatting DL/I Calls for Language Interfaces	77
Application Programming for Assembler Language	78
Format	78
Parameters	79
Example DL/I Call Formats	80
Application Programming for C Language	80
Format	80
Parameters	81
I/O Area	83
Example DL/I Call Formats	83
Application Programming for COBOL	83
Format	83
Parameters	84
Example DL/I Call Formats	85
Application Programming for Pascal	86
Format	86
Parameters	87
Example DL/I Call Formats	88
Application Programming for PL/I	88
Format	88
Parameters	89
Example DL/I Call Formats	90
Relationship of Calls to PCBs	91
Specifying the I/O PCB Mask	92
Specifying the DB PCB Mask	95
Specifying the AIB Mask	98
Specifying the AIB Mask for ODBA Applications	99
AIB Examples	101
Specifying the UIB (CICS Online Programs Only)	102
Specifying the I/O Areas	105
Segment Search Arguments	106
SSA Coding Rules	106
SSA Coding Restrictions	107
SSA Coding Formats	107
GSAM Data Areas	111
GSAM DB PCB Masks	111
GSAM RSAs	111
The AIBTDLI Interface	111
Overview	111
Defining Storage for the AIB	112
Specifying the Language Specific Entry Point	112
Assembler Language	112
C Language	113
COBOL	113
Pascal	114
PL/I	114
Interface Considerations	114
PCB Lists	115
Format of a PCB List	115
Format of a GPSB PCB List	115
PCB Summary	115
The AERTLDI interface	116
Overview	116
Defining Storage for the AIB	117
Language Environment	117

The CEETDLI interface to IMS. . . . .	118
LANG= Option on PSBGEN for PL/I Compatibility with Language Environment . . . . .	118
Special DL/I Situations. . . . .	118
Application Program Scheduling against HALDBs. . . . .	119
Mixed-Language Programming . . . . .	119
Language Environment Routine Retention . . . . .	120
Extended Addressing Capabilities of MVS/ESA . . . . .	120
Preloaded Programs . . . . .	120
<b>Chapter 4. Writing DL/I Calls for Database Management . . . . .</b>	<b>121</b>
CIMS Call . . . . .	121
Format . . . . .	121
Parameters. . . . .	122
Usage. . . . .	122
CLSE Call . . . . .	123
Format . . . . .	123
Parameters. . . . .	123
Usage. . . . .	123
DEQ Call . . . . .	123
Format (Full Function). . . . .	123
Format (Fast Path DEDB) . . . . .	124
Parameters. . . . .	124
Usage. . . . .	124
Restrictions. . . . .	125
DLET Call . . . . .	125
Format . . . . .	125
Parameters. . . . .	125
Usage. . . . .	126
FLD Call. . . . .	126
Format . . . . .	126
Parameters. . . . .	126
Usage. . . . .	127
FSAs . . . . .	127
GN/GHN Call . . . . .	128
Format . . . . .	128
Parameters. . . . .	129
Usage, Get Next (GN). . . . .	130
Usage, Get Hold Next (GHN) . . . . .	131
Usage, HDAM, PHDAM, or DEDB Database with GN . . . . .	132
Restriction . . . . .	132
GNP/GHNP Call . . . . .	132
Format . . . . .	133
Parameters. . . . .	133
Usage, Get Next in Parent (GNP) . . . . .	133
Usage, Get Hold Next in Parent (GHNP) . . . . .	135
GU/GHU Call . . . . .	135
Format . . . . .	135
Parameters. . . . .	136
Usage, Get Unique (GU). . . . .	136
Usage, Get Hold Unique (GHU) . . . . .	137
Restriction . . . . .	138
ISRT Call . . . . .	138
Format . . . . .	138
Parameters. . . . .	138
Usage. . . . .	139

OPEN Call . . . . .	141
Format . . . . .	141
Parameters . . . . .	141
Usage. . . . .	142
POS Call . . . . .	142
Format . . . . .	142
Parameters . . . . .	142
Usage. . . . .	145
Restrictions. . . . .	145
REPL Call . . . . .	145
Format . . . . .	145
Parameters . . . . .	146
Usage. . . . .	146
<b>Chapter 5. Writing DL/I Calls for System Services . . . . .</b>	<b>149</b>
APSB Call . . . . .	150
Format . . . . .	150
Parameters . . . . .	150
Usage. . . . .	150
CHKP (Basic) Call . . . . .	150
Format . . . . .	151
Parameters . . . . .	151
Usage. . . . .	151
CHKP (Symbolic) Call . . . . .	151
Format . . . . .	151
Parameters . . . . .	152
Usage. . . . .	153
Restrictions. . . . .	153
DPSB Call . . . . .	153
Format . . . . .	153
Parameters . . . . .	153
Usage. . . . .	153
GMSG Call . . . . .	154
Format . . . . .	154
Parameters . . . . .	154
Usage. . . . .	155
Restrictions. . . . .	156
GSCD Call . . . . .	156
Format . . . . .	156
Parameters . . . . .	156
Usage. . . . .	157
Restriction . . . . .	157
ICMD Call . . . . .	157
Format . . . . .	157
Parameters . . . . .	157
Usage. . . . .	158
Restrictions. . . . .	159
INIT Call . . . . .	159
Format . . . . .	159
Parameters . . . . .	159
Usage. . . . .	159
Restrictions. . . . .	163
INQY Call . . . . .	163
Format . . . . .	164
Parameters . . . . .	164
Usage. . . . .	164



Restrictions.	169
LOG Call	169
Format	169
Parameters.	169
Usage.	170
Restrictions.	171
PCB Call (CICS Online Programs Only)	171
Format	171
Parameters.	171
Usage.	171
Restrictions.	172
RCMD Call	172
Format	172
Parameters.	172
Usage.	173
Restrictions.	173
ROLB Call	173
Format	173
Parameters.	173
Restrictions.	174
ROLL Call	174
Format	174
Parameters.	174
Usage.	174
Restriction	174
ROLS Call	175
Format	175
Parameters.	175
Usage.	175
Restrictions.	176
SETS/SETU Call.	176
Format	176
Parameters.	176
Usage.	177
Restrictions.	177
SNAP Call	177
Format	177
Parameters.	177
Usage.	180
Restrictions.	180
STAT Call	180
Format	180
Parameters.	181
Usage.	182
Restrictions.	182
SYNC Call	182
Format	182
Parameters.	183
Usage.	183
Restrictions.	183
TERM Call (CICS Online Programs Only)	183
Format	183
Usage.	183
Restrictions.	184
XRST Call	184
Format	184

Parameters . . . . .	184
Usage. . . . .	185
Restrictions. . . . .	187
<b>Chapter 6. Monitoring Your Position in the Database . . . . .</b>	<b>189</b>
Understanding Current Position in the Database . . . . .	189
Position after Retrieval Calls . . . . .	190
Position after DLET. . . . .	191
Position after REPL. . . . .	193
Position after ISRT . . . . .	193
Current Position after Unsuccessful Calls. . . . .	194
Position after an Unsuccessful DLET or REPL Call . . . . .	194
Position after an Unsuccessful Retrieval or ISRT Call . . . . .	195
<b>Chapter 7. Multiple Qualification Statements . . . . .</b>	<b>199</b>
Overview of Multiple Qualification Statements . . . . .	199
Example using Multiple Qualification Statements . . . . .	200
Multiple Qualification Statements for HDAM, PHDAM, or DEDB . . . . .	201
<b>Chapter 8. Multiple Processing . . . . .</b>	<b>203</b>
Multiple Positioning . . . . .	203
Advantages of Using Multiple Positioning . . . . .	206
How Multiple Positioning Affects Your Program. . . . .	206
Resetting Position with Multiple Positioning . . . . .	208
Using Multiple DB PCBs . . . . .	208
<b>Chapter 9. Secondary Indexing and Logical Relationships . . . . .</b>	<b>211</b>
How Secondary Indexing Affects Your Program . . . . .	211
SSAs with Secondary Indexes . . . . .	211
Multiple Qualification Statements with Secondary Indexes . . . . .	212
What DL/I Returns with a Secondary Index . . . . .	214
Status Codes for Secondary Indexes . . . . .	214
Processing Segments in Logical Relationships . . . . .	214
How Logical Relationships Affect Your Programming . . . . .	216
Status Codes for Logical Relationships . . . . .	217
<b>Chapter 10. Processing GSAM Databases . . . . .</b>	<b>219</b>
Accessing GSAM Databases . . . . .	219
PCB Masks for GSAM Databases . . . . .	219
Retrieving and Inserting GSAM Records . . . . .	221
Explicitly Opening and Closing a GSAM Database . . . . .	222
GSAM Record Formats . . . . .	222
GSAM I/O Areas . . . . .	223
GSAM Status Codes . . . . .	223
Symbolic CHKP and XRST with GSAM . . . . .	224
GSAM Coding Considerations . . . . .	224
Origin of GSAM Data Set Characteristics . . . . .	225
DD Statement DISP Parameter for GSAM Data Sets . . . . .	226
Using Extended Checkpoint Restart for GSAM Data Sets . . . . .	226
Use of Concatenated Data Sets by GSAM . . . . .	227
Suggested Method for Specifying GSAM Data Set Attributes . . . . .	227
DLI or DBB Region Types and GSAM . . . . .	227
<b>Chapter 11. Processing Fast Path Databases . . . . .</b>	<b>229</b>
Fast Path Database Calls . . . . .	229
MSDBs and DEDBs: Overview . . . . .	230

MSDBs . . . . .	230
DEDBs . . . . .	231
Processing MSDBs and DEDBs . . . . .	231
Updating Segments in an MSDB or DEDB: REPL, DLET, ISRT, and FLD	231
Commit-Point Processing in MSDBs and DEDBs . . . . .	235
VSO Considerations . . . . .	236
Data Locking for MSDBs and DEDBs . . . . .	236
Restrictions on Using Calls for MSDBs . . . . .	237
Processing DEDBs (IMS, CICS with DBCTL) . . . . .	238
Processing DEDBs with Subset Pointers . . . . .	238
Retrieving Location with the POS Call (for DEDB Only) . . . . .	242
Commit-Point Processing in a DEDB . . . . .	245
Crossing a UOW Boundary (P Processing Option) . . . . .	245
Crossing the UOW Boundary (H Processing Option) . . . . .	245
Data Locking . . . . .	246
Restrictions on Using Calls for DEDBs. . . . .	246
Direct Dependent Segments . . . . .	246
Sequential Dependent Segments. . . . .	247
Fast Path Coding Considerations. . . . .	247
<b>Chapter 12. Recovering Databases and Maintaining Database Integrity</b>	249
Issuing Checkpoints . . . . .	249
Restarting Your Program and Checking for Position . . . . .	249
Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)	250
Backing Out to a Prior Commit Point: ROLL, ROLB, and ROLS . . . . .	250
Backing Out to an Intermediate Backout Point: SETS, SETU, and ROLS	254
Reserving Segments for the Exclusive Use of Your Program. . . . .	256
Resource Lock Management . . . . .	257



---

## Chapter 1. How Application Programs Work with the IMS Database Manager

Your application program uses Data Language I (DL/I) to communicate with the IMS Database Manager (IMS DB). This section provides an overview of the database management process.

### **In this Chapter:**

- “Application Program Environments”
- “The Application Programming Interface” on page 9
- “Getting Started with DL/I” on page 12
- “Getting Started with DL/I (for CICS Online Users)” on page 13
- “Getting Started with DL/I using the ODBA Interface” on page 15
- “DL/I Calls” on page 16
- “High Availability Large Databases” on page 18
- “Sample Hierarchies” on page 19
- “SSA Overview” on page 24
- “Command Codes” on page 28
- “IVP Sample Application Program” on page 45

Application programming techniques and the application programming interface are explained here as they apply to the IMS DB.

### **Related Reading:**

- If your installation uses the IMS Transaction Manager (IMS TM), refer to the *IMS Version 9: Application Programming: Transaction Manager* for information on transaction management functions.
- Information on DL/I EXEC commands is in the *IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS*.

---

## Application Program Environments

Your application program can execute in different IMS environments. The three online environments are DB/DC, DBCTL, and DCCTL.

The two batch environments are:

- DB batch, which is generated from DB/DC and DBCTL class system generations.
- TM batch, which is generated from DCCTL class system generations.

This book describes applications that execute in DB/DC, DBCTL, and DB Batch environments.

**Related Reading:** For information on DCCTL and TM Batch environments, see *IMS Version 9: Application Programming: Transaction Manager*.

---

## The Application Programming Interface

The information in this section provides an overview of the role your application program plays in the IMS DB system. The IMS environments described within this subtopic are DB/DC, DBCTL, and DB Batch.

**Related Reading:** For additional system-level information on IMS DB, see *IMS Version 9: Administration Guide: Database Manager*.

## The DB/DC Environment

The DB/DC environment is composed of a control region and dependent regions. The dependent regions might include Message Processing Program (MPP), Batch Message Processing Program (BMP), and IMS Fast Path Program (IFP) regions. Application programs can reside in any dependent region. Messages and database calls from application programs, and messages and commands from terminals are sent to and processed by the control region. The IMS control region retrieves and processes the needed information and returns it to the application program or terminal. However, only application programs residing in the BMP region can access GSAM databases. These calls are not processed by the IMS control region, but are passed directly through the BMP region. Figure 2 shows how an application program can be positioned in a DB/DC environment.

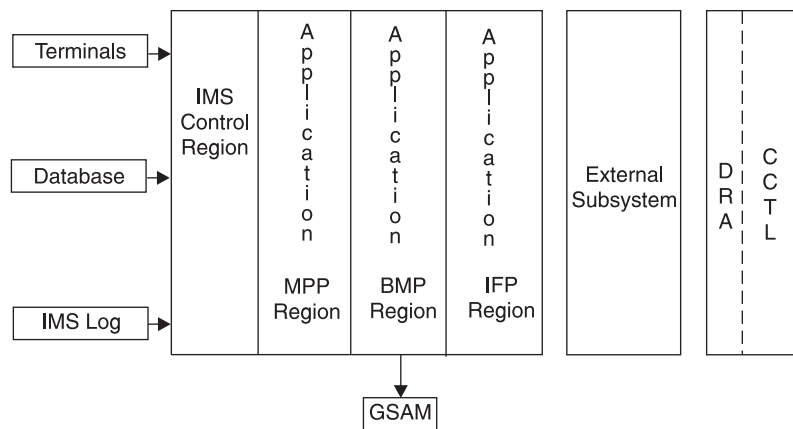


Figure 2. Application View of DB/DC Environment

The online environment can be used to access other types of external subsystems using the External Subsystem Attach facility (ESAF). It lets application programs obtain data from external subsystems such as DB2.

All DL/I database management calls and most system service calls are supported in DB/DC. For more information on calls supported in DB/DC, see Chapter 15, “Summary of DM and System Service Calls,” on page 297.

The IMS DB portion of the IMS DB/DC environment can be used separately to provide database management capabilities for coordinator controllers (CCTLs). The IMS DB portion is called the DBCTL environment.

**Related Reading:** For more information on IMS DB/DC environments, refer to *IMS Version 9: Administration Guide: Database Manager* or *IMS Version 9: Administration Guide: System*.

## The DBCTL Environment

DBCTL behaves in the same manner as IMS DB in a DB/DC environment, but it does not support user terminals, a master terminal, or message handling. One interface to DBCTL is the Database Resource Adapter (DRA). The DRA can be used in two scenarios:

- If communications and transaction management services are needed, they are provided by a Coordinator Controller (CCTL). A CCTL consists of the DRA and a transaction management subsystem, such as CICS. The DRA resides in the same address space as the transaction management subsystem, thus enabling communication between the IMS DB environment and the “connected” transaction management subsystem.

The CCTL handles message traffic, schedules applications outside the IMS DB environment, and passes database calls through the DRA to IMS DB. IMS DB processes the DL/I call and returns the information to the CCTL through the DRA. See Figure 3 for an illustration of the IMS DB environment with a CCTL.

- A z/OS application program can use the Open Database Access (ODBA) callable interface to access databases managed by an IMS DB subsystem. Internally, ODBA uses the DRA to establish a connection to the IMS subsystem specified by the IMSID.

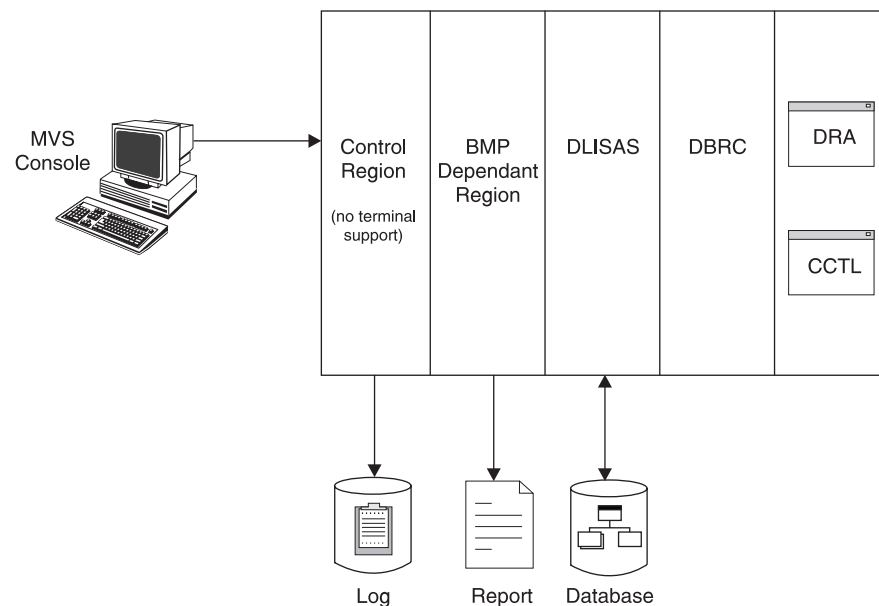


Figure 3. Application View of DBCTL Environment

Most DL/I database management calls and system service calls are supported in DBCTL. They are listed in Chapter 15, “Summary of DM and System Service Calls,” on page 297. IMS application programs in the DBCTL environment can run in non-message-driven BMP regions. Application programs for DBCTL are the same as IMS DB application programs. However, DBCTL application programs cannot issue DL/I calls for communications or access MSDBs. DBCTL BMPs can access DL/I, DEDB, and GSAM databases.

The DBCTL environment can also be used to attach to an external subsystem, such as DB2, using the External Subsystem Attach facility (ESAF). The DBCTL environment’s ability to attach to external subsystems provides a BMP access to DB2 databases. Application programs running under a CCTL do not have access to external subsystems or GSAM through the DRA interface.

**Related Reading:** For more information on IMS DBCTL environments, refer to *IMS Version 9: Administration Guide: Database Manager* and *IMS Version 9: Administration Guide: System*.

## The DB Batch Environment

DB Batch is the batch environment that is generated during DB/DC or DBCTL system generations. The DB Batch environment has a single address space that contains both IMS code and the application program. DB Batch application programs have access to DL/I and GSAM databases.

**Related Reading:** For more information on IMS DB Batch environments, refer to *IMS Version 9: Administration Guide: Database Manager* and *IMS Version 9: Administration Guide: System*.

---

## Getting Started with DL/I

The information in this section applies to all application programs that run in IMS. The main elements in an IMS application program consist of the following:

- Program entry
- Program Communication Block (PCB) or Application Interface Block (AIB) definition
- I/O area definition
- DL/I calls
- Program termination

Figure 4 shows how these elements relate to each other. The numbers on the right in Figure 4 refer to the notes that follow.

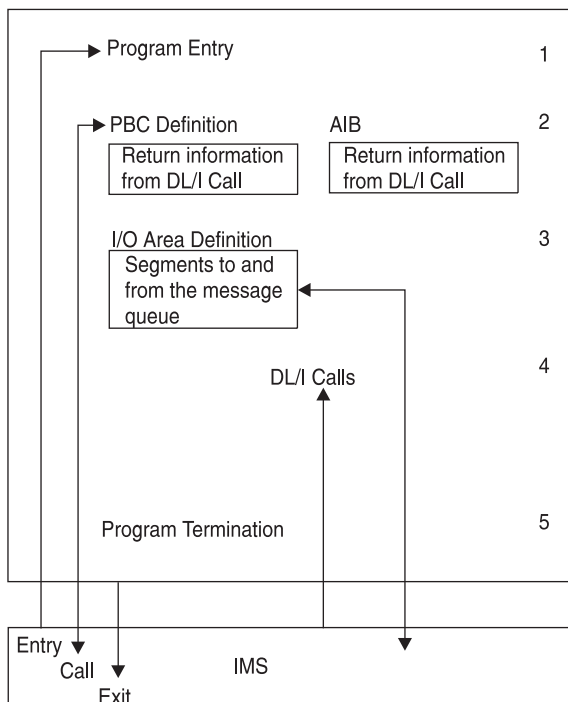


Figure 4. DL/I Program Elements

### Notes to Figure 4:

1. **Program entry.** IMS passes control to the application program with a list of associated PCBs.
2. **PCB or AIB.** IMS describes the results of each DL/I call using the AIBTDLI interface in the application interface block (AIB) and, when applicable, the



program communication block (PCB). To find the results of a DL/I call, your program must use the PCB that is referenced in the call. To find the results of the call using the AIBTDLI interface, your program must use the AIB.

Your application program can use the PCB address that is returned in the AIB to find the results of the call. To use the PCB, the program defines a mask of the PCB and can then reference the PCB after each call to determine the success or failure of the call. An application program cannot change the fields in a PCB; it can only check the PCB to determine what happened when the call was completed.

3. **Input/output (I/O) area.** IMS passes segments to and from the program in the program's I/O area.
4. **DL/I calls.** The program issues DL/I calls to perform the requested function.
5. **Program Termination.** The program returns control to IMS DB when it has finished processing. In a batch program, your program can set the return code and pass it to the next step in the job.

**Recommendation:** If your program does not use the return code in this way, it is a good idea to set it to 0 as a programming convention. Your program can use the return code for this same purpose in BMPs. (MPPs cannot pass return codes.)

---

## Getting Started with DL/I (for CICS Online Users)

The information here applies to call-level CICS programs that use Database Control (DBCTL). DBCTL provides a database subsystem that runs in its own address space and gives one or more CICS/ESA® systems access to IMS DL/I full-function databases and DEDBs.

Figure 5 on page 14 shows the structure of a call-level CICS online program. A few differences exist between CICS online and batch programs. For example, in a CICS online program, you must issue a call to schedule a program specification block (PSB). See Figure 5 on page 14 notes for a description of each program element depicted in the figure.

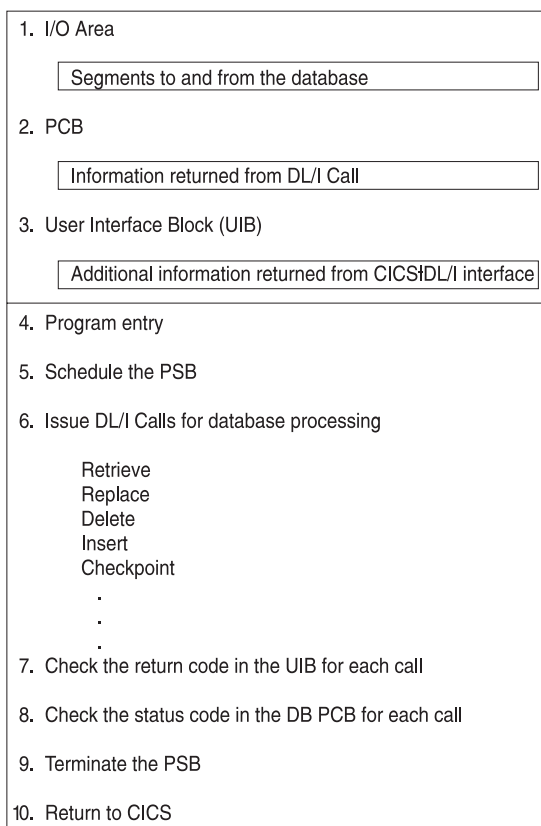


Figure 5. The Structure of a Call-Level CICS Online Program

#### Notes to Figure 5:

- I/O area.** IMS passes segments to and from the program in the program's I/O area.
- PCB.** IMS describes the results of each DL/I call in the database PCB mask.
- User Interface Block (UIB).** The UIB provides the program with addresses of the PCBs and return codes from the CICS-DL/I interface.  
The horizontal line between number 3 (User Interface Block (UIB)) and number 4 (Program entry) in Figure 5, represents the end of the declarations section and the start of the executable code section of the program.
- Program entry.** CICS passes control to the application program during program entry. Do not use an ENTRY statement as you would in a batch program.
- Schedule the PSB.** This identifies the PSB your program is to use and passes the address of the UIB to your program.
- Issue DL/I Calls.** You issue DL/I calls to read and update the database.
- Check the return code in the UIB.** You should check the return code after issuing any DL/I call for database processing, including the PCB or TERM call. Do this before checking the status code in the PCB.
- Check the status code in the PCB.** You should check the status code after issuing any DL/I call for database processing. The code gives you the results of your DL/I call.
- Terminate the PSB.** This terminates the PSB and commits database changes. PSB termination is optional, and if it is not done, the PSB is released when your program returns control to CICS.

10. **Return to CICS.** This returns control to either CICS or the linking program. If control is returned to CICS, database changes are committed, and the PSB is terminated.

---

## Getting Started with DL/I using the ODBA Interface

This section applies to z/OS application programs that use database resources that are managed by IMS DB. Open Database Access (ODBA) is an interface that enables the z/OS application programs to access IMS DL/I full-function databases and data entry databases (DEDBs).

This section has three parts.

- “Common Logic Flow for SRMS and MRMS” describes the common logic flow for both single resource manager scenarios (SRMS) and multiple resource manager scenarios (MRMS).
- “Logic Flow for SRMS” describes how the programmer commits changes for SRMS. The programmer commits changes in step one of this part.
- “Logic Flow for MRMS” on page 16 describes how the programmer commits changes in MRMS. The programmer can commit changes anytime after step 1 of this part.

### Common Logic Flow for SRMS and MRMS

The common logic flow for SRMS and MRMS is described in steps one through nine. The logic flow differences for SRMS and MRMS are described in “Logic Flow for SRMS” and “Logic Flow for MRMS” on page 16.

1. I/O area. IMS passes segments to and from the application program in the its I/O area.
2. PCB. IMS describes the results of each DL/I call in the database PCB mask.
3. Application Interface Block (AIB). The AIB provides the program with addresses of the PCBs and return codes from the ODBA-DL/I interface.
4. Program entry. Obtain and initialize the AIB.
5. Initialize the ODBA interface.
6. Schedule the PSB. This step identifies the PSB that your program is to use and also provides a place for IMS to keep internal tokens.
7. Issue DL/I Calls. You issue DL/I calls to read and update the database. The following calls are available:
  - Retrieve
  - Replace
  - Delete
  - Insert
8. Check the return code in the AIB. You should check the return code after issuing any DL/I call for database processing. Do this before checking the status code in the PCB.
9. Check the status code in the PCB. If the AIB return code indicates (Return Code X'900'), then you should check the status code after issuing any DL/I call for database processing. The code gives you the results of your DL/I call.

### Logic Flow for SRMS

1. Commit database changes. No DL/I calls, including system service calls such as LOG or STAT, can be made between the commit and the termination of the DPSB.

2. Terminate the DPSB.
3. Terminate the ODBA interface.
4. Return to environment that initialized the application program.

## Logic Flow for MRMS

1. Terminate the PSB.
2. Terminate the ODBA interface.
3. Commit changes.
4. Return to environment that initialized the application program.

---

## DL/I Calls

A DL/I call consists of a call statement and a list of parameters. The parameters provide information that IMS needs to execute the call. This information consists of the call function, the name of the data structure that IMS uses for the call, the data area in the program into which IMS returns data, and any condition that the retrieved data must meet.

You can issue calls to perform database management (DB calls) and to obtain IMS DB system service (system service calls):

## DB Call Functions

The DL/I calls for database management are:

<b>CLSE</b>	GSAM Close
<b>DEQ</b>	Dequeue
<b>DLET</b>	Delete
<b>FLD</b>	Field
<b>GHN</b>	Get Hold Next
<b>GHNP</b>	Get Hold Next in Parent
<b>GHU</b>	Get Hold Unique
<b>GN</b>	Get Next
<b>GNP</b>	Get Next in Parent
<b>GU</b>	Get Unique
<b>ISRT</b>	Insert
<b>OPEN</b>	GSAM Open
<b>POS</b>	Position
<b>REPL</b>	Replace

## System Service Call Functions

The DL/I calls for system service are:

<b>APSB</b>	Allocate PSB
<b>CHKP</b>	Basic Checkpoint
<b>CHKP</b>	Symbolic Checkpoint
<b>CIMS</b>	ODBA Function

<b>DPSB</b>	Deallocate PSB
<b>GMSG</b>	Get Message
<b>GSCD</b>	Get Address of System Contents Directory
<b>ICMD</b>	Issue Command
<b>INIT</b>	Initialize
<b>INQY</b>	Inquiry
<b>LOG</b>	Log
<b>PCB</b>	Specify and Schedule a PCB
<b>RCMD</b>	Retrieve Command
<b>ROLB</b>	Roll Back
<b>ROLL</b>	Roll
<b>ROLS</b>	Roll Back to SETS
<b>SETS</b>	Set a Backout Point
<b>SETU</b>	SET Unconditional
<b>SNAP</b>	Collects diagnostic information
<b>STAT</b>	Statistics
<b>SYNC</b>	Synchronization
<b>TERM</b>	Terminate
<b>XRST</b>	Extended Restart

**Related Reading:**

- DL/I calls are described in detail in Chapter 4, “Writing DL/I Calls for Database Management,” on page 121 and Chapter 5, “Writing DL/I Calls for System Services,” on page 149.
- Reference tables for the calls appear in Chapter 15, “Summary of DM and System Service Calls,” on page 297 and “System Service Call Summary” on page 298.

**Status Codes, Return Codes, and Reason Codes**

To give information about the results of each call, IMS places a two-character status code in the PCB after each IMS call your program issues. Your program should check the status code after every IMS call. If it does not check the status code, the program might continue processing even though the previous call caused an error.

The status codes your program should test for are those that indicate **exceptional but valid conditions**. *IMS Version 9: Messages and Codes, Volume 1* lists the status codes that may be returned by each call type and indicates the level of success for each call. Your program should check for status codes which indicate the call was successful, such as blanks. If IMS returns a status code that you did not expect, your program should branch to an error routine.

Information for your calls is supplied in status codes that are returned in the PCB, return and reason codes that are returned in the AIB, or both.

## Exceptional Conditions

Some status codes do not mean that your call was successful or unsuccessful; they just give information about the results of the call. Your program uses this information to determine what to do next. The meanings of these status codes depend on the call.

In a typical program, status codes that you should test for apply to the get calls. Some status codes indicate exceptional conditions for other calls, and you should provide routines other than error routines for these situations. For example, AH means that a required SSA is missing, and AT means that the user I/O area is too long.

## High Availability Large Databases

You need to be aware that the feedback on data availability at PSB schedule time shows the availability of only the High Availability Large Database (HALDB) master, not of the HALDB partitions. However, the error settings for data unavailability of a HALDB partition are the same as those of a non-HALDB database, namely status code 'BA' or pseudo abend U3303.

Also note that logical child segments cannot be loaded into a HALDB PHDAM or PHIDAM database. Logical child segments must be inserted later in an update run. Any attempt to load a logical child segment in either a PHDAM or PHIDAM database results in status code LF.

## Error Routines

If your program detects an error after checking for blanks and exceptional conditions in the status code, it should branch to an error routine and print as much information as possible about the error before terminating. Determining which call was being executed when the error occurred, what parameters were on the IMS call, and the contents of the PCB will be helpful in understanding the error. Print the status code to help with problem determination.

Two kinds of errors can occur in your program: programming errors and system or I/O errors. Programming errors, are usually your responsibility to find and fix. These errors are caused by things like an invalid parameter, an invalid call, or an I/O area that is too long. System or I/O errors are usually resolved by the system programmer or the equivalent specialist at your installation.

Because every application program should have an error routine, and because each installation has its own ways of finding and debugging program errors, you probably have your own standard error routines.

## DL/I and Your Application Program

When an application program call is issued to IMS, control passes to IMS from the application program. Standard subroutine linkage and parameter lists link IMS to your application program. After control is passed, IMS examines the input parameters, which perform the request functions.

## DBDs and PSBs

Application programs can communicate with databases without being aware of the physical location of the data they possess. To do this, database descriptors (DBDs) and program specification blocks (PSBs) are used.

A DBD describes the content and hierarchic structure of the physical or logical database. DBDs also supply information to IMS to help in locating segments.

A PSB specifies the database segments an application program can access and the functions it can perform on the data, such as read only, update, or delete. Because an application program can access multiple databases, PSBs are composed of one or more program control blocks (PCBs). The PSB describes the way a database is viewed by your application program.

Figure 6 shows the normal relationship between application programs, PSBs, PCBs, DBDs, and databases.

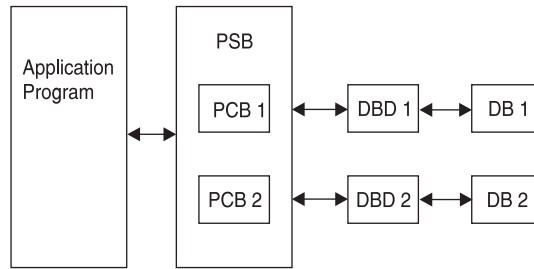


Figure 6. Normal Relationship between Programs, PSBs, PCBs, DBDs, and Databases

Figure 7 shows concurrent processing, which uses multiple PCBs for the same database.

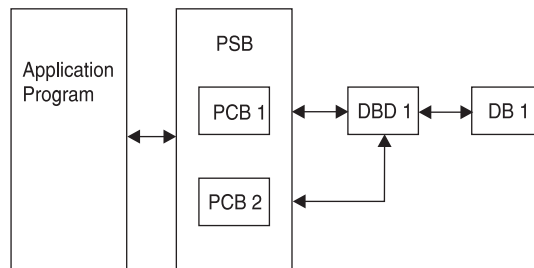


Figure 7. Relationship between Programs and Multiple PCBs (Concurrent Processing)

## SSAs and Command Codes

Segment search arguments (SSAs) are specific arguments that describe the segments that you are looking for. Calls can be qualified or unqualified. A *qualified* call uses SSAs to form a complete path to the segment. An *unqualified* call does not use SSAs at all. See “SSA Overview” on page 24 for more information.

Command codes enhance your application program by requesting a number of IMS DB functions that save programming and processing time. Table 14 on page 28 shows the command codes used for application programming.

---

## Sample Hierarchies

The examples in this information use the medical hierarchy shown in Figure 8 on page 20 and the bank hierarchies shown in Table 8 on page 22, Table 9 on page 23, and Table 10 on page 23. The medical hierarchy is used with full-function databases and Fast Path DEDBs. The bank hierarchies are an example of an

application program used with main storage databases (MSDBs). To understand these examples, familiarize yourself with the hierarchies and segments that each hierarchy contains.

## Medical Database Example

The medical database shown in Figure 8 contains information that a medical clinic keeps about its patients.

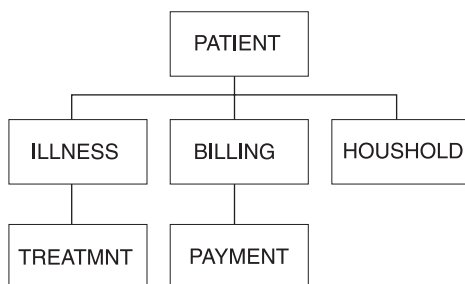


Figure 8. Medical Hierarchy

The tables that follow show the layouts of each segment in the hierarchy. The segment's field contents are in the first row of each table. The number below each field contents is the length in bytes that has been defined for that field.

- **PATIENT Segment**

Table 2 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 of their patient numbers. The lowest patient number in the database is 00001 and the highest is 10500.

Table 2. PATIENT Segment

Field Contents	PATNO	NAME	ADDR
Bytes	5	10	30

- **ILLNESS Segment**

Table 3 on page 21 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 3. ILLNESS Segment

Field Contents	ILLDATE	ILLNAME
Bytes	8	10

• **TREATMNT Segment**

Table 4 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 4. TREATMNT Segment

Field Contents	DATE	MEDICINE	QUANTITY	DOCTOR
Bytes	8	10	4	10

• **BILLING Segment**

Table 5 shows the BILLING segment. It has only one field: the amount of the current bill. BILLING has no key field.

Table 5. BILLING Segment

Field Contents	BILLING
Bytes	6

• **PAYMENT Segment**

Table 6 shows the PAYMENT segment. It has only one field: the amount of payments for the month. The PAYMENT segment has no key field.

Table 6. PAYMENT Segment

Field Contents	PAYMENT
Bytes	6

• **HOUSHOLD Segment**

Table 7 on page 22 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 HOUSEHOLD segment's key field is RELNAME.

Table 7. HOUSEHOLD Segment

Field Contents	RELNAME	RELATN
Bytes	10	8

## Bank Account 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. "Related MSDBs" and "Nonrelated MSDBs" on page 23 illustrate the differences between these types of databases. Additional information on how related and nonrelated MSDBs differ is provided under "Processing MSDBs and DEBDBs" on page 231.

### 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:

<b>TELLERID</b>	A two-character code that identifies the teller
<b>TRANCNT</b>	The number of transactions the teller has processed
<b>TELLBAL</b>	The balance for the teller

Table 8 shows what the segment for this type of application program looks like.

Table 8. 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:

<b>BRANCHNO</b>	The identification number for the branch
<b>TOTAL</b>	The bank branch's current balance
<b>TRANCNT</b>	The number of transactions for the branch on that day
<b>DEPBAL</b>	The deposit balance, giving the total dollar amount of deposits for the branch
<b>WTHBAL</b>	The withdrawal balance, giving the dollar amount of the withdrawals for the branch

Table 9 shows what the branch summary segment looks like in a dynamic related MSDB.

*Table 9. 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:

<b>ACCNTNO</b>	The account number
<b>BRANCH</b>	The name of the branch where the account is
<b>TRANCNT</b>	The number of transactions for this account this month
<b>BALANCE</b>	The current balance

Table 10 shows what the account segment in a nonrelated MSDB application program looks like.

*Table 10. 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.

---

## SSA Overview

Segment Search Arguments (SSAs) specify information for IMS to use in processing a DL/I call. A DL/I call with one or more SSAs is a *qualified call*, and a DL/I call without SSAs is an *unqualified call*.

### Definitions:

#### **Unqualified SSA**

Contains only a segment name.

#### **Qualified SSA**

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 SSAs 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.

Table 11 shows the structure of a qualified SSA. Table 12 on page 27 shows the structure of an unqualified SSA using command codes. Finally, Table 13 on page 28 shows the structure of a qualified SSA that uses 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:

```
PATIENTbb
```

## 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. Table 11 shows the structure of a qualified SSA.

Table 11. Qualified SSA Structure

Seg Name	(	Fld Name	R.O.	Fld Value	)
8	1	8	2	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 Table 11) 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:

```
PATIENTb(PATNObbb=b10460)
```

The qualification statement is enclosed in parentheses. The first field contains the name of the field (F1d Name in Table 11 on page 24) 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
  - =b
  - b=
  - EQ
- Greater than, represented as
  - >b
  - b>
  - GT
- Less than, represented as
  - <b
  - b<
  - 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 (F1d Value in Table 11 on page 24) contains the value that you want IMS to use as the comparative value. The length of F1d Value must be the same length as the field specified by F1d 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.

**Related Reading:** For more information on multiple qualification statements, see Chapter 7, “Multiple Qualification Statements,” on page 199.

### Using the Sequence Field 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. Figure 9 on page 26 shows a segment with a noncontiguous sequence field.

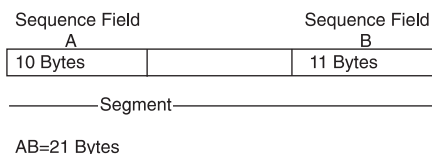


Figure 9. Segment with a Noncontiguous Sequence Field

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 9: Administration Guide: Database Manager*.

## Guidelines for Using SSAs

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 more information on secondary indexing, see Chapter 9, “Secondary Indexing and Logical Relationships,” on page 211.

**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;PATIENTb(PATN0bbb=b09000)
```

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.

## 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 "Command Codes" on page 28.

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*.

**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 Figure 10.

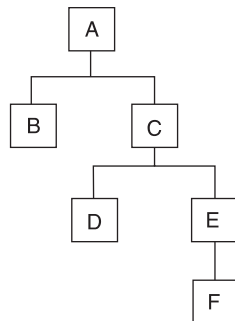


Figure 10. D Command Code Example

The call function and the SSAs for the call look like this:

```
GU  Abbbbbbb*D
      Cbbbbbbb*D
      Ebbbbbbb*D
      Fbbbbbbb
```

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 Table 12.

Table 12. Unqualified SSA with Command Code

Seg Name	*	Cmd Code	b
8	1	Variable	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 Table 13.

Table 13. Qualified SSA with Command Code

Seg Name	*	Cmd Code	(	Fld Name	R.O.	Fld Value	)
8	1	Variable	1	8	2	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. This process is described in detail in "Processing DEDBs with Subset Pointers" on page 238.

---

## Command Codes

This section describes the command codes used for DL/I calls and it is divided into two subtopics.

- "General Command Codes for DL/I Calls" on page 29 covers the C, D, F, L, N, P, Q, U, V, and null command codes, which are used with full-function databases and DEDBs.
- "DEDB Command Codes for DL/I" on page 39 covers the M, R, S, W, and Z command codes, which are used only with DEDBs.

See Table 14 for all the command codes and their usage.

**Restriction:** Command codes cannot be used by MSDB calls.

Table 14. Command Codes for DL/I Calls

Command Code	Usage
C	Supplies concatenated key in SSA
D	Retrieves or inserts a sequence of segments
F	Starts search with first occurrence
L	Locates last occurrence
M <sup>1</sup>	Moves subset pointer forward to the next segment
N	Prevents replacement of a segment on a path call
P	Establishes parentage of present level
Q	Enqueues segment
R <sup>1</sup>	Retrieves first segment in the subset
S <sup>1</sup>	Sets subset pointer unconditionally
U	Maintains current position
V	Maintains current position at present level and higher
W <sup>1</sup>	Sets subset pointer conditionally



Table 14. Command Codes for DL/I Calls (continued)

Command Code	Usage
Z <sup>1</sup>	Sets subset pointer to 0
- (null)	Reserves storage positions for program command codes in SSA

**Note:**

1. This command code is used only with DEDBs.

## General Command Codes for DL/I Calls

This section has descriptions and examples for the C, D, F, L, N, P, Q, U, V, and null command codes.

### The C Command Code

You can use the C command code to indicate to IMS that (instead of supplying a qualification statement) you are supplying the segment's concatenated key as a means of identifying it. You can use either the C command code or a qualification statement, but not both.

You can use the C command code for all Get calls and for the ISRT call. When you code the concatenated key, enclose it in parentheses following the \*C, and place it in the same position that would otherwise contain the qualification statement.

**Example:** Suppose you wanted to satisfy the following request:

Did Joan Carter visit the clinic on March 3, 1993? Her patient number is 07755.

The PATIENT segment's key field is the patient number, and the ILLNESS segment's key field is the date field, so the concatenated key is 0775519930303. This number is comprised of four digits for the year, followed by two digits for both the month and the day. You issue a GU call with the following SSA to satisfy the request:

```
GU ILLNESSb*C(0775519930303)
```

Using the C command code is sometimes more convenient than a qualification statement because it is easier to use the concatenated key than to move each part of the qualification statement to the SSA area during program execution. Using the segment's concatenated key is the equivalent of giving all the SSAs in the path to the segment qualified on their keys.

**Example:** Suppose that you wanted to answer the following request:

What treatment did Joan Carter, patient number 07755, receive on March 3, 1993?

Using qualification statements, you would specify the following SSAs with a GU call:

```
GU PATIENTb(PATN0bbbEQ07755)
   ILLNESSb(ILLDATEbEQ19930303)
   TREATMNTb
```

Using a C command code, you can satisfy the previous request by specifying the following SSAs on a GU call:

```
GU    ILLNESSb*C(0775519930303)
      TREATMNTb
```

If you need to qualify a segment by using a field other than the key field, use a qualification statement instead of the C command code.

Only one SSA with a concatenated key is allowed for each call. To return segments to your program in the path to the segment specified by the concatenated key, you can use unqualified SSAs containing the D command code.

**Example:** If you want to return the PATIENT segment for Joan Carter to your I/O area, in addition to the ILLNESS segment, use the following call:

```
GU    PATIENTb*Db
      ILLNESSb*C(0775519930303)
```

You can use the C command code with the object segment for a Get call, but not for an ISRT call. The object segment for an ISRT call must be unqualified.

### The D Command Code

You can use the D command code to retrieve or insert a sequence of segments in a hierarchic path with one call rather than retrieving or inserting each segment with a separate call. A call that uses the D command code is called a *path call*.

For your program to use the D command code, the P processing option must be specified in the PCB, unless your program uses command code D when processing DEDBs.

**Related Reading:** For more information on using the P processing option, see the description of PSB generation in *IMS Version 9: Utilities Reference: System*.

**Retrieving a Sequence of Segments:** When you use the D command code with retrieval calls, IMS places the segments in your I/O area. The segments in the I/O area are placed one after the other, left to right, starting with the first SSA you supplied. To have IMS return each segment in the path, you must include the D command code in each SSA. You can, however, include intervening SSAs without the D command code. You do not need to include the D command code on the last segment in the path, because IMS always returns the last segment in the path to your I/O area.

The D command code has no effect on IMS's retrieval logic. The only thing it does is cause each segment to be moved to your I/O area. The segment name in the PCB is the lowest-level segment that is retrieved or the last level that is satisfied in the call in the case of a GE (not-found) status code. Higher-level segments with the D command code are placed in the I/O area.

If IMS is unable to find the lowest segment your program has requested, it returns a GE (not-found) status code, just as it does if your program does not use the D command code and IMS is unable to find the segment your program has requested. This is true even if IMS reaches the end of the database before finding the lowest segment your program requested. If IMS reaches the end of the database without satisfying any levels of a path call, it returns a GB (end of database) status code. However, if IMS returns one or more segments to your I/O area (new segments for which there was no current position at the start of the current call), and if IMS is unable to find the lowest requested segment, IMS returns a GE status code, even if it has reached the end of the database.

The advantages of using the D command code are significant even if your program is not sure that it will need the dependent segment returned by D. For example, suppose that after examining the dependent segment, your program still needs to use it. Using the D command, your program has the segment if you need it, and your program is not required to issue another call for the segment.

**Example:** As an example of the D command code, suppose your program has this request:

Compute the balance due for each of the clinic's patients by subtracting the payments received from the amount billed; print bills to be mailed to each patient.

To process this request for each patient, your program needs to know the patient's name and address, what the charges are for the patient, and the amount of payment the patient has made. Issue this call until your program receives a GE status code indicating that no more patient segments exist:

```
GN    PATIENTb*D
      BILLINGb*D
      PAYMENTbb
```

Each time you issue this call, your I/O area contains the patient segment, the billing segment, and the payment segment for a particular person.

**Inserting a Sequence of Segments:** With ISRT calls, your program can use the D command code to insert a path of segments simultaneously. Your program need not include D for each SSA in the path. Your program just specifies D on the first segment that you want IMS to insert. IMS inserts the segments in the path that follow.

**Example:** Suppose your program has the following request:

Judy Jennison visited the clinic for the first time. Add a record that includes PATIENT, ILLNESS, and TREATMNT segments.

After building the segments in your I/O area, issue an ISRT call with the following SSAs:

```
ISRT  PATIENTb*Db
      ILLNESSbb
      TREATMNTb
```

Not only is the PATIENT segment added, but the segments following the PATIENT segment, ILLNESS and TREATMNT, are also added to the database.

You cannot use the D command code to insert segments if a logical child segment in the path exists.

## The F Command Code

You can use the F command code to start the search with the first occurrence of a certain segment type or to insert a new segment as the first occurrence in a chain of segments.

**Retrieving a Segment as the First Occurrence:** You can use the F command code for GN and GNP calls. Using it with GU calls is redundant (and is disregarded) because GU calls can already back up in the database. When you use F, you indicate that you want the search to start with the first occurrence of the segment type you indicate under its parent in attempting to satisfy this level of the call.

You can use the F command code for GN and GNP calls to back up in the database. You can back up to the first occurrence of the segment type that has current position, or you can back up to a segment type that is before current position in the hierarchy.

**Restriction:** The parent of the segment that you are backing up from must be in the same hierarchic path as the segment you are backing up to. IMS disregards F when you supply it at the root level or with a GU or GHU.

The search must start with the first occurrence of the segment type that you indicate under the parent. When the search at that level is satisfied, that level is treated as though a new occurrence of a segment has satisfied the search. This is true even when the segment that satisfies an SSA where F command code is specified is the same segment occurrence on which DL/I was positioned before the call was processed.

When a new segment occurrence satisfies an SSA, the position of all dependent segments is reset. New searches for dependent segments then start with the first occurrence of that segment type under its parent.

**Inserting a Segment as the First Occurrence:** When you use F with an ISRT call, you are indicating that you want IMS to insert the segment you have supplied as the first segment occurrence of its segment type. Use F with segments that have either no key at all or a non unique key, and that have HERE specified on the RULES operand of the SEGM statement in the DBD. If you specify HERE in the DBD, the F command code overrides this, and IMS inserts the new segment occurrence as the first occurrence of that segment type.

Using the F command code to override the RULES specification on the DBD applies only to the path (either logical or physical) that you are using to access the segment for the ISRT call. For example, if you are using the physical path to access the segment, the command code applies to the physical path but not to the logical path. For clarification of using command codes with the RULES specification, ask the database administrator at your installation.

**Example:** Suppose that you specified RULES=HERE in the DBD for the TREATMNT segment. You want to satisfy the following request:

Mary Martin visited the clinic today and visited a number of different doctors. Add the TREATMNT segment for Dr. Smith as the first TREATMNT segment for the most recent illness.

First you build a TREATMNT segment in your I/O area:

```
19930302ESEDRIXbbb0040SMITHbbbb
```

Then you issue an ISRT call with the following SSAs. This adds a new occurrence of the TREATMNT segment as the first occurrence of the TREATMNT segment type among those with equal keys.

```
ISRT  PATIENTb(PATNObbb=b06439)
      ILLNESSb*L
      TREATMNT*F
```

This example applies to HDAM or PHDAM root segments and to dependent segments for any type of database.

## The L Command Code

You can use the L command code to retrieve the last occurrence of a particular segment type or to insert a segment as the last occurrence of a segment type.

**Retrieving a Segment as the Last Occurrence:** The L command code indicates that you want to retrieve the last segment occurrence that satisfies the SSA, or that you want to insert the segment occurrence you are supplying as the last occurrence of that segment type. Like F, L simplifies your programming because you can go directly to the last occurrence of a segment type without having to examine the previous occurrences with program logic, if you know that it is the last segment occurrence that you want. L can be used with GU or GHU, because IMS normally returns the first occurrence when you use a GU call. IMS disregards L at the root level.

Using L with GU, GN, and GNP indicates to IMS that you want the last occurrence of the segment type that satisfies the qualification you have provided. The qualification is the segment type or the qualification statement of the SSA. If you have supplied just the segment type (an unqualified SSA), IMS retrieves the last occurrence of this segment type under its parent.

**Example:** Suppose you have this request using the medical hierarchy:

What was the illness that brought Jennifer Thompson, patient number 10345, to the clinic most recently?

In this example, assume that RULES=LAST is specified in the DBD for the database on ILLNESS. Issue this call to retrieve this information:

```
GU    PATIENTb(PATN0bbb=b10345)
      ILLNESSb*L
```

The first SSA gives IMS the number of the particular patient. The second SSA asks for the last occurrence (in this case, the first occurrence chronologically) of the ILLNESS segment for this patient.

**Inserting a Segment as the Last Occurrence:** Use L with ISRT only when the segment has no key or a non unique key, and the insert rule for the segment is either FIRST or HERE. Using the L command code overrides both FIRST and HERE for HDAM or PHDAM root segments and dependent segments in any type of database.

Using the L command code to override the RULES specification on the DBD applies only to the path (either logical or physical) that you are using to access the segment for the ISRT call. For example, if you are using the physical path to access the segment, the command code applies to the physical path but not to the logical path. For clarification of using command codes with the RULES specification, ask your database administrator.

## The N Command Code

The N command code prevents you from replacing a segment on a path call. In conjunction with the D command code, it lets the application program to process multiple segments using one call. Alone, the D command code retrieves a path of segments in your I/O area. With the N command code, the D command code lets you distinguish which segments you want to replace.

**Example:** The following code only replaces the TREATMNT segment.

```
GHU  PATIENT*D(PATNObbb=b06439)
      ILLNESSb*D(ILLDATEb=19930301)
      TREATMNT
REPL PATIENT*N(PATNObbb=b06439)
      ILLNESSb*N(ILLDATEb=19930301)
      TREATMNT
```

**Restriction:** If you use D and N command codes together, IMS retrieves the segment but does not replace it.

The N command code applies only to REPL calls, and IMS ignores it if you include the code in any other call.

### The P Command Code

Ordinarily, IMS sets parentage at the level of the lowest segment that is accessed during a call. To set parentage at a higher level, you can use the P command code in a GU, GN, or GNP call.

The parentage that you set with P works just like the parentage that IMS sets: it remains in effect for subsequent GNP calls, and is not affected by ISRT, DLET, or REPL calls. It is only affected by GNP if you use the P command code in the GNP call. Parentage is canceled by a subsequent GU, GHU, GN, or GHN.

Use the P command code at only one level of the call. If you mistakenly use P in multiple levels of a call, IMS sets parentage at the lowest level of the call that includes P.

If IMS cannot fully satisfy the call that uses P (for example, IMS returns a GE status code), but the level that includes P is satisfied, P is still valid. If IMS cannot fully satisfy the call including the level that contains P, IMS does not set any parentage. You would receive a GP (no parentage established) if you then issued a GNP.

If you use P with a GNP call, IMS processes the GNP call with the parentage that was already set by preceding calls. IMS then resets parentage with the parentage you specified using P after processing the GNP call.

**Example:** If you want to send a current bill to all of the patients seen during the month, the determining value is in the ILLNESS segment. You want to look at only patients whose ILLNESS segments have dates after the first of the month. For patients who have been to the clinic during the month, you need to look at their addresses and the amount of charges in the BILLING segment so that you can print a bill. For this example, assume the date is March 31, 1993. Issue these two calls to process this information:

```
GN  PATIENTb*PD
    ILLNESSb(ILLDATEb>=19930301)
GNP BILLINGbbb
```

After you locate a patient who has been to the clinic during the month, you issue the GNP call to retrieve that patient's BILLING segment. Then you repeat the GN call to find each patient who has been to the clinic during the month, until IMS returns a GB status code.

### The Q Command Code

Use the Q command code if you want to prevent another program from updating a segment until your program reaches a commit point. The Q command code tells IMS that your application program needs to work with a segment and that no other tasks can be allowed to modify the segment until the program has finished. This

means that you can retrieve segments using the Q command code, then retrieve them again later, knowing that they have not been altered by another program. (You should be aware, however, that reserving segments for the exclusive use of your program can affect system performance.)

You can use the Q command code in batch programs in a data-sharing environment and in CICS and IMS online programs. IMS ignores Q in non-data sharing batch programs.

**Limiting the Number of Database Calls:** For full function, before you use the Q command code in your program, you must specify a MAXQ value during PSBGEN. This establishes the maximum number of database calls (with Q command codes) that you can make between sync points.

**Related Reading:** For information on PSBGEN, see *IMS Version 9: Utilities Reference: System*.

Fast Path does not support the MAXQ parameter. Consequently in Fast Path, you can issue as many database calls with Q command codes as you want.

**Using Segment Lock Class:** For full function, when you use the Q command code to retrieve a segment, you specify the letter Q followed by a letter (A-J), designating the lock class of that segment (for example, QA). If the lock class is not a letter (A-J), IMS returns the status code GL.

Fast Path supports the Q command code alone, without a letter designating the lock class. However, for consistency between Fast Path and full function, Fast Path treats the Q command code as a 2-byte string, where the second byte must be a letter (A-J). If the second byte is not a letter (A-J), IMS returns the status code AJ.

**Example:** Suppose a customer wants to place an order for items 1, 2, and 3, but only if 50 item 1's, 75 item 2's, and 100 item 3's are available. Before placing this order, the program must examine all three item segments to determine whether an adequate number of each item is available. You do not want other application programs to change any of the segments until your program has determined this and, if possible, placed the order.

To process this request for full function, your program uses the Q command code when it issues the Get calls for the item segments. When you use the Q command code in the SSA, you assign a lock class immediately following the command code in the SSA.

```
GU  PART X
    ITEM 1  *QA
GU  PART X
    ITEM 2  *QA
GU  PART X
    ITEM 3  *QA
```

**Exception:** For Fast Path, the second byte of the lock class is not interpreted as lock class 'A'.

After retrieving the item segments, your program can examine them to determine whether an adequate number of each item are on hand to place the order. Assume 100 of each item are on hand. Your program then places the order and updates the database accordingly. To update the segment, your program issues a GHU call for each segment and follows it immediately with a REPL call:



```

GHU  ITEM 1
REPL ITEM 1 with the value 50
GHU  ITEM 2
REPL ITEM 2 with the value 25
GHU  ITEM 3
REPL ITEM 3 with the value 0

```

**Using the DEQ Call with the Q Command Code:** When you use the Q command code and the DEQ call, you reserve and release segments.

For full function, to issue a DEQ call against an I/O PCB to release a segment, you place the letter designating the segment's lock class in the first byte of an I/O area. Then, you issue the DEQ call with the name of the I/O area that contains the letter.

A DEDB DEQ call is issued against a DEDB PCB. Because Fast Path does not support lock class, a DEDBDEQ call does not require that a lock class be specified in the I/O area.

**Restriction:** The EXEC DL/I interface does not support DEDB DEQ calls, because EXEC DL/I disallows a PCB for DEQ calls.

**Retrieving Segments with Full-Function DEQ Calls:** The DEQ call releases all segments that are retrieved using the Q command code, except:

- Segments modified by your program, until your program reaches a commit point
- Segments required to keep your position in the hierarchy, until your program moves to another database record
- A class of segments that has been locked again as another class

If your program only reads segments, it can release them by issuing a DEQ call. If your program does not issue a DEQ call, IMS releases the reserved segments when your program reaches a commit point. By releasing them with a DEQ call before your program reaches a commit point, you make them available to other programs more quickly.

**Retrieving Buffers with Fast Path DEQ Calls:** DEQ calls cause Fast Path to release a buffer that satisfies one of the following conditions:

- The buffer has not been modified, or the buffer does not protect a valid root position.
- The buffer has been protected by a Q command code.

Fast Path returns an FW status code when no buffers can be released for a DEQ call.

Any CI locking or segment-level locking performed with a Q command code is protected from other application programs until a DEQ call is issued or a commit point is reached.

**Considerations for Root and Dependent Segments (Full Function Only):** If you use the Q command code on a root segment, other programs in which the PCB does not have update capability can access the database record. Programs in which the PCB has update capability cannot access any of the segments in that database record. If you use the Q command code on a dependent segment, other programs can read the segment using one of the Get calls without the hold. If your program accesses shared databases, and if any of the segments in that block are reserved with the Q command code, application programs in other IMS systems



cannot update anything in that block. The Q command code does not hold segments from one step of a conversation to another.

**Related Reading:** For more information on the relationship between the Q command code and the DEQ call, see “Reserving Segments for the Exclusive Use of Your Program” on page 256.

### The U Command Code

As IMS satisfies each level in a retrieval or ISRT call, a position on the segment occurrence that satisfies that level is established.

The U command code prevents position from being moved from a segment during a search of its hierarchic dependents. If the segment has a unique sequence field, using this code is equivalent to qualifying the SSA so that it is equal to the current value of the key field. When a call is being satisfied, if position is moved to a level above that at which the U code was issued, the code has no effect for the segment type whose parent changed position.

U is especially useful when unkeyed dependents or non unique keyed segments are being processed. The position on a specific occurrence of an unkeyed or non unique keyed segment can be held by using this code.

**Example:** Suppose you want to find out about the illness that brought a patient named Mary Warren to the clinic most recently, and about the treatments she received for that illness. Figure 11 shows the PATIENT, ILLNESS, and TREATMNT segments for Mary Warren.

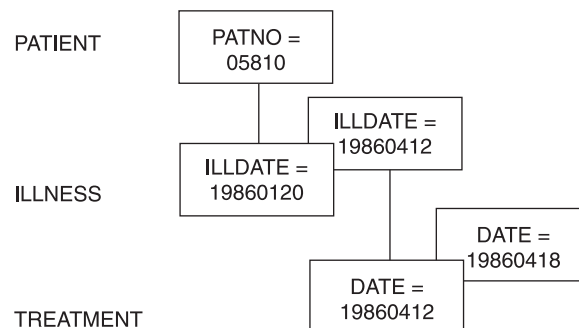


Figure 11. U Command Code Example

To retrieve this information, retrieve the first ILLNESS segment and the TREATMNT segments associated with that ILLNESS segment. To retrieve the most recent ILLNESS segment, you can issue the following GU call:

```

GU  PATIENTb(PATNObbb=b05810
    ILLNESSb*L
  
```

After this call, IMS establishes a position at the root level on the PATIENT segment with the key 05810 and on the last ILLNESS segment. Because other ILLNESS segments with the key 19860412 may exist, you can think of this one as the most recent ILLNESS segment. You might want to retrieve the TREATMNT segment occurrences that are associated with that ILLNESS segment. You can do this by issuing the GN call below with the U command code:

```

GN  PATIENTb*U
    ILLNESSb*U
    TREATMNT
  
```

In this example, the U command code indicates to IMS that you want only TREATMNT segments that are dependents of the ILLNESS and PATIENT segments on which IMS has established position. Issuing the above GN call the first time retrieves the TREATMNT segment with the key of 19860412. Issuing the GN call the second time retrieves the TREATMNT segment with the key 19860418. If you issue the call a third time, IMS returns a not-found status code. The U command code tells IMS that, if it does not find a segment that satisfies the lower qualification under this parent, it cannot continue looking under other parents. If the U command code was not in the PATIENT SSA, the third GN call causes IMS to move forward at the root level in an attempt to satisfy the call. If you supply a U command code for a qualified SSA, IMS ignores the U.

If used in conjunction with command code F or L, the U command code is disregarded at the level and all lower levels of SSAs for that call.

### The V Command Code

Using the V command code on an SSA is similar to using a U command code in that SSA and all preceding SSAs. Specifying the V command code for a segment level tells IMS that you want to use the position that is established at that level and above as qualification for the call.

Using the V command code is analogous to qualifying your request with a qualified SSA that specifies the current IMS position.

**Example:** Suppose that you wanted to answer the following request:

Did Joan Carter, patient number 07755, receive any treatment on March 3, 1993?

Using qualified SSAs, specify the following call:

```
GU    PATIENTb(PATNObbb=b07755)
      ILLNESSb(ILLDATEb=19930303)
      TREATMNT
```

If you have position established on the PATIENT segment for patient number 07755 and on the ILLNESS segment for March 3, 1993, you can use your position to retrieve the TREATMNT segments in which you are interested. You do this by specifying the V command code as follows:

```
GN    PATIENTbb
      ILLNESSb*V
      TREATMNT
```

Using the V command code for a call is like establishing parentage and issuing a subsequent GNP call, except that the V command code sets the parentage for the call it is used with, not for subsequent calls. For example, to satisfy the previous request, you could have set parentage at the ILLNESS segment level and issued a GNP to retrieve any TREATMNT segments under that parent. With the V command code, you specify that you want the ILLNESS segment to be used as parentage for that call.

You can specify the V command code for any parent segment. If you use the V command code with a qualified SSA, it is ignored for that level and for any higher level that contains a qualified SSA.

### The NULL Command Code

The null command code (-) enables you to reserve one or more positions in an SSA in which a program can store command codes, if they are needed during program execution.

**Example:** Reserve position for two command codes as follows:

```
GU      PATIENTb*--(PATN0bbb=b07755)
        ILLNESSb(ILLDATEb=19930303)
        TREATMNT
```

Using the null command code lets you use the same set of SSAs for more than one purpose. However, dynamically modifying SSAs makes debugging more difficult.

## DEDB Command Codes for DL/I

The M, R, S, W, and Z command codes are only used with a DEDB. The examples in this subtopic are based on the following scenario.

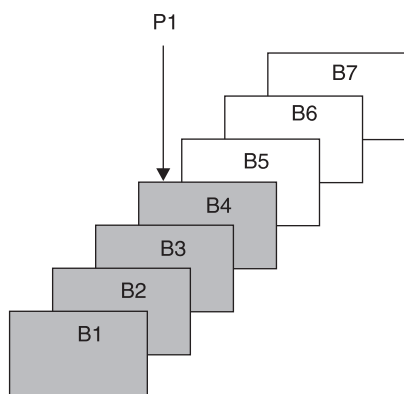
### Sample Application Program

The examples in this section are based on one sample application program—the recording of banking transactions for a passbook (savings account) 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 0. An application program that updates the database later will be able to tell that no unposted transactions exist. Figure 12 summarizes the processing that is performed when the passbook is unavailable and when it is available.

1. When the passbook is unavailable . . .



2. When the passbook is available . . .

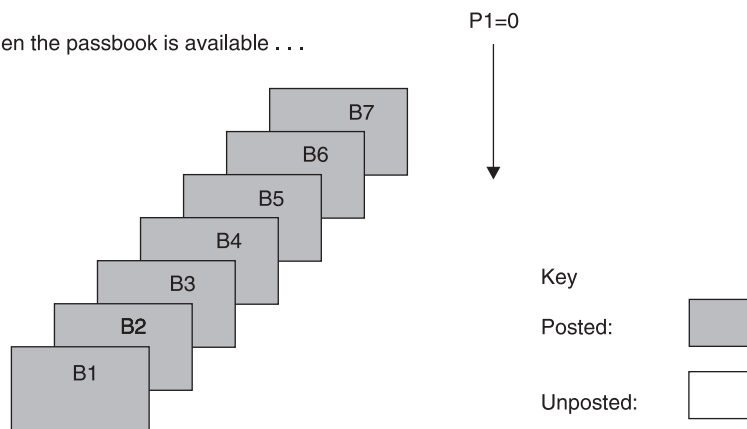


Figure 12. Processing for the Passbook Example

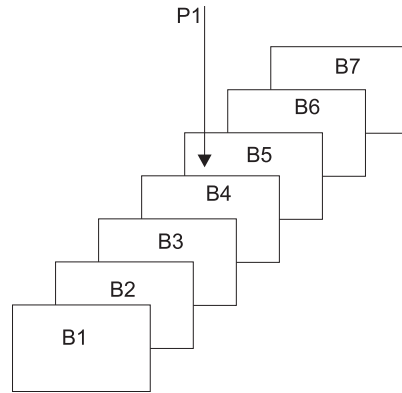
### The M Command Code

To move the subset pointer forward to the next segment after your current position, your program issues a call with the M command code. Using the passbook account example, suppose that you want to post some, but not all, of the transactions, and that you want the subset pointer to be set to the first unposted transaction. The following command sets subset pointer 1 to segment B6, as shown in Figure 13.

```
GU   Abbbbbbb(AKEYbbb
      Bbbbbbbb*R1M1
```

If the current segment is the last in the chain, and you use an M command code, IMS sets the pointer to 0.

Before the call:



After the call:

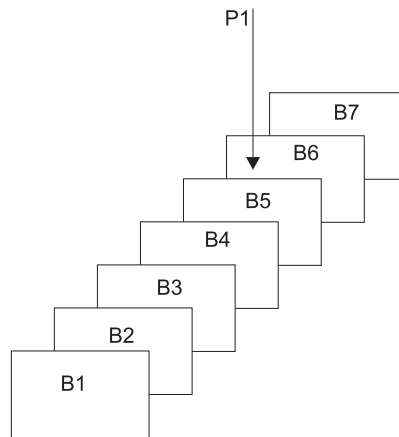


Figure 13. Moving the Subset Pointer to the Next Segment after Your Current Position

### The R Command Code

To retrieve the first segment occurrence in the subset, your program issues a Get call with the R command code. The R command code does not set or move the pointer. It indicates to IMS that you want to establish position on the first segment occurrence in the subset. The R command code is like the F command code, except that the R command code applies to the subset instead of to the entire segment chain.

Using the passbook account example, suppose that Bob Emery visits the bank and brings his passbook; you want to post all of the unposted transactions. Because subset pointer 1 was previously set to the first unposted transaction, your program uses the following call to retrieve that transaction:

```
GU      Abbbbbbb (AKEYbbbb=bA1)
        Bbbbbbbb*R1
```

As shown by Figure 14 on page 42, this call retrieves segment B5. To continue processing segments in the chain, you can issue GN calls as you would if you were not using subset pointers.

If the subset does not exist (subset pointer 1 has been set to 0), IMS returns a GE status code, and your position in the database will be immediately following the last

segment in the chain. Using the passbook example, the GE status code tells you that no unposted transactions exist.

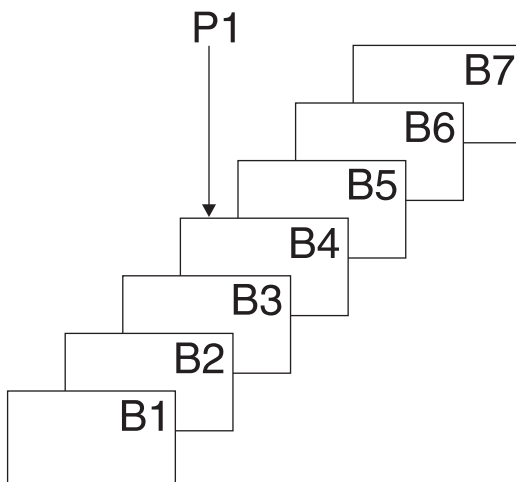


Figure 14. Retrieving the First Segment in a Chain of Segments

You can specify only one R command code for each SSA. If you use more than one R in an SSA, IMS returns an AJ status code to your program.

You can use R with other command codes, except F and Q. Other command codes in an SSA take effect after the R command code has been processed, and after position has been successfully established on the first segment in the subset. If you use the L and R command codes together, the last segment in the segment chain is retrieved. (If the subset pointer that was specified with the R command code, IMS returns a GE status code instead of the last segment in the segment chain.) Do not use the R and F command codes together. If you do, you will receive an AJ status code. The R command code overrides all insert rules, including LAST.

### The S Command Code

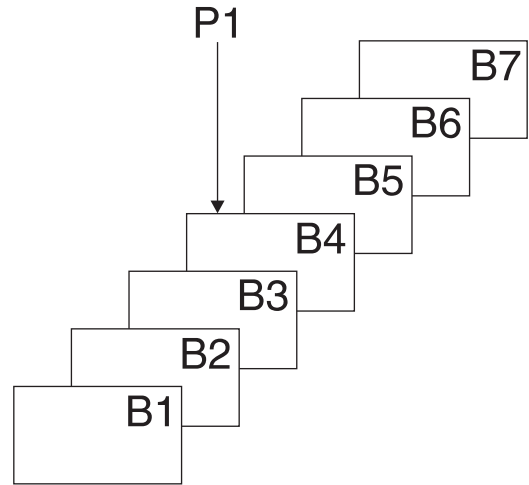
To set a subset pointer unconditionally, regardless of whether it is already set, your program issues a call with the S command code. “The W Command Code” on page 43 describes how to set a subset pointer only if it is not already set. When your program issues a call that includes the S command code, IMS sets the pointer to your current position.

**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:

```
GU      Abbbbbbb(AKEYbbbb=bB1)
        Bbbbbbbb*R1
GN      Bbbbbbbb*S1
```

After you issue this call, instead of pointing to segment B5, subset pointer 1 points to segment B6, as shown in Figure 15 on page 43.

Before the command:



After the command:

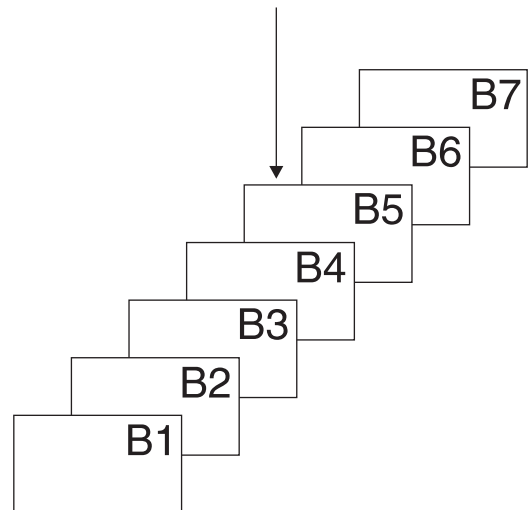


Figure 15. Unconditionally Setting the Subset Pointer to Your Current Position

**The W Command Code**

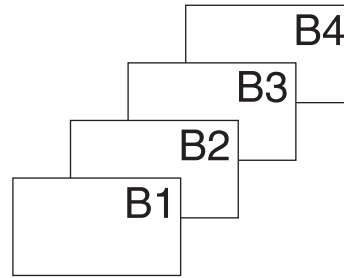
Like the S command code, the W command code sets the subset pointer conditionally. Unlike the S command code, the W command code updates the subset pointer only if the subset pointer is not already set to a segment.

**Example:** Using the passbook example, suppose 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 later, when you post the transactions, you can immediately access the first one. The following call sets the subset pointer to the transaction you are inserting if it is the first unposted one.

```
ISRT      Abbbbbbb (AKEYbbbb=bA1)
          Bbbbbbbb*W1
```

As shown by Figure 16, this call sets subset pointer 1 to segment B5. If unposted transactions already exist, the subset pointer is not changed.

Before the command:



After the command:

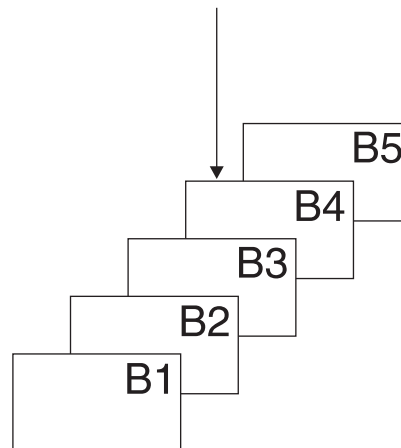


Figure 16. Conditionally Setting the Subset Pointer to Your Current Position

**The Z Command Code**

The Z command code sets the value of the subset pointer to 0. After your program issues a call with the Z command code, the pointer is no longer set to a segment, and 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 0.)

**Example:** Using the passbook example, suppose that you used the R command code to retrieve the first unposted transaction. You then process the chain of segments, posting the transactions. After posting the transactions and inserting any new ones into the chain, use the Z command code to set the subset pointer to 0 as shown in the following call:

```
ISRT      Abbbbbbb (AKEYbbbb=bA1)
          Bbbbbbbb*Z1
```

After this call, subset pointer 1 is set to 0, which indicates to a program that subsequently updates the database that no unposted transactions exist.



---

## IVP Sample Application Program

The IVP sample application program is a very simple phone book application. Each of the application programs performs the same add, change, delete, and display functions. The source for the IVP Sample Application is in the IMS.SDFSISRC (SMP/E target) library. Two programs are provided in several different languages. The two programs are:

**DFSIVA3** A Conversational MPP that accesses an HDAM/VSAM database. Transaction input and output is through MFS screens.

**DFSIVA6** A Batch or BMP program that accesses a HIDAM/OSAM database. The program uses GSAM to receive its transaction input and to display its transaction output.

These programs are fully installed and executed by the IVP.

The IMS EXEC library also includes the REXX exec named DFSSUT04 EXEC. Use this exec to process any unexpected return codes or status codes.

**Related Reading:** A full description of the IVP Sample Application is in the *IMS Version 9: Installation Volume 1: Installation Verification*.



---

## Chapter 2. Writing Your Application Programs

This section contains suggestions for writing a more efficient application program, a checklist of coding considerations, and skeleton programs in assembler language, C language, COBOL, Pascal, and PL/I.

### **In this Chapter:**

- “Programming Guidelines”
- “Coding DL/I Calls and Data Areas” on page 48
- “Preparing to Run Your CICS DL/I Call Program” on page 49
- “Sample Programs” on page 49

---

### **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.
- 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.

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 Reading:** For more information about segment locking, see “Reserving Segments for the Exclusive Use of Your Program” on page 256.

---

## 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. Before you start coding, make sure you have the information described in this section.

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 is 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 link-editing your CICS online program, see the description of installing application programs in *CICS/ESA 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 IBM COBOL for z/OS™ & VM or VS COBOL II, see *CICS/ESA Application Programming Guide*.

---

## Sample Programs

This section contains sample programs written in assembler language, C language, COBOL, Pascal, and PL/I. The programs are examples of how to code DL/I calls and data areas. They are not complete programs. Before running them, you must modify them to suit the requirements of your installation.

## Coding a Batch Program in Assembler Language

Figure 17 on page 51 is a skeleton program that shows how the parts of an IMS program written in assembler language fit together. 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.

```

PGMSTART C
NOTES
*           EQUATE REGISTERS
1
*   USEAGE OF REGISTERS
R1     EQU  1      ORIGINAL PCBLIST ADDRESS
R2     EQU  2      PCBLIST ADDRESS1
R5     EQU  5      PCB ADDRESSSS
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  CL1')'
*

```

Figure 17. Sample Assembler Language Program (Part 1 of 2)

```

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
DBPCBRVS 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

```

## ASSEMBLER LANGUAGE INTERFACE

12

Figure 17. Sample Assembler Language Program (Part 2 of 2)

**Notes to Figure 17 on page 51:**

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.

12. The IMS-supplied language interface module (DFSLI000) must be link-edited with the compiled assembler language program.

**Related Reading:** For more information on installing CICS application programs, see *CICS/MVS Installation Guide*.

## Coding a CICS Online Program in Assembler Language

Figure 18 on page 54 is a skeleton program in assembler language. It 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.

```

PGMSTART DSECT
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'ASMP SB'
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

```

Figure 18. Sample Call-Level Assembler Language Program (CICS Online) (Part 1 of 2)

```

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

```

Figure 18. Sample Call-Level Assembler Language Program (CICS Online) (Part 2 of 2)

#### Notes to the example:

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, which is expanded as shown under “Specifying the UIB (CICS Online Programs Only)” on page 102.
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 SSAs, define each part of the SSA separately so that the program can modify the SSA’s fields.

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.

## Coding a Batch Program in C Language

Figure 19 on page 57 is a skeleton batch program that shows you how the parts of an IMS program that is written in C fit together. The numbers to the right of the program refer to the notes that follow the program.

```

#pragma runopts(env(IMS),plist(IMS))
#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;

```

Figure 19. Sample C Language Program (Part 1 of 2)

```

/*                                     */
/*      Initialize the qualifier        */
/*                                     */

    sprintf(qual_ssa,
            "%-8.8s(%-8.8s%2.2s%-6.6s)",
            "ROOT", "KEY", "=", "vvvvv");
7
/*                                     */
/*      Main part of C batch program   */
/*                                     */
/*                                     */
    ctdli(func_GU, detail_PCB,
          &det_seg_io_area,qual_ssa);
9
    ctdli(func_GHU, mast_PCB,
          &mast_seg_io_area,qual_ssa);
10
    ctdli(func_GHN, mast_PCB,
          &mast_seg_io_area);
11
}
12

C LANGUAGE INTERFACE | 13

```

Figure 19. Sample C Language Program (Part 2 of 2)

#### Notes to Figure 19:

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, **COMPAT=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, **COMPAT=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.
13. IMS provides a language interface module (DFSLI000), which gives a common interface to IMS. This module must be made available to the application program at link-edit time.

## Coding a Batch Program in COBOL

The program in Figure 20 is a skeleton batch program that shows you how the parts of an IMS program, written in COBOL, fit together. 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.



```

ENVIRONMENT DIVISION.

1
.
.
DATA DIVISION.
WORKING-STORAGE SECTION.
  77 FUNC-GU          PICTURE XXXX VALUE 'GU  '.
  77 FUNC-GHU        PICTURE XXXX VALUE 'GHU  '.
  77 FUNC-GN         PICTURE XXXX VALUE 'GHN  '.
  77 FUNC-GHN        PICTURE XXXX VALUE 'GHN  '.
  77 FUNC-GNP        PICTURE XXXX VALUE 'GNP  '.
  77 FUNC-GHNP       PICTURE XXXX VALUE 'GHNP'.
  77 FUNC-REPL       PICTURE XXXX VALUE 'REPL'.
  77 FUNC-ISRT       PICTURE XXXX VALUE 'ISRT'.
  77 FUNC-DLET       PICTURE XXXX VALUE 'DLET'.
  77 COUNT           PICTURE S9(5)VALUE +4 COMPUTATIONAL.
  01 UNQUAL-SSA.
    02 SEG-NAME      PICTURE X(08) VALUE '      '.
    02 FILLER        PICTURE X      VALUE '  '.
2
  01 QUAL-SSA-MAST.
    02 SEG-NAME-M    PICTURE X(08) VALUE 'ROOTMAST'.
    02 BEGIN-PAREN-M PICTURE X      VALUE '('.
    02 KEY-NAME-M    PICTURE X(08) VALUE 'KEYMAST '.
    02 REL-OPER-M    PICTURE X(02) VALUE '='.
    02 KEY-VALUE-M   PICTURE X(06) VALUE 'vvvvvv'.
    02 END-PAREN-M   PICTURE X      VALUE ')'.
3
  01 QUAL-SSA-DET.
    02 SEG-NAME-D    PICTURE X(08) VALUE 'ROOTDET '.
    02 BEGIN-PAREN-D PICTURE X      VALUE '('.
    02 KEY-NAME-D    PICTURE X(08) VALUE 'KEYDET  '.
    02 REL-OPER-D    PICTURE X(02) VALUE '='.
    02 KEY-VALUE-D   PICTURE X(06) VALUE 'vvvvvv'.
    02 END-PAREN-D   PICTURE X      VALUE ')'.
  01 DET-SEG-IN.
    02 —
    02 —
  01 MAST-SEG-IN.
4
    02 —
    02 —
LINKAGE SECTION.
  01 IO-PCB.
    02 FILLER          PICTURE X(10).
    02 IO-STAT-CODE    PICTURE XX.
    02 FILLER          PICTURE X(20).
  01 DB-PCB-MAST.
    02 MAST-DBD-NAME   PICTURE X(8).
    02 MAST-SEG-LEVEL PICTURE XX.
5
    02 MAST-STAT-CODE  PICTURE XX.
    02 MAST-PROC-OPT  PICTURE XXXX.
    02 FILLER          PICTURE S9(5) COMPUTATIONAL.
    02 MAST-SEG-NAME   PICTURE X(8).
    02 MAST-LEN-KFB    PICTURE S9(5) COMPUTATIONAL.
    02 MAST-NU-SENSESEG PICTURE S9(5) COMPUTATIONAL.
    02 MAST-KEY-FB     PICTURE X—X.
  01 DB-PCB-DETAIL.
    02 DET-DBD-NAME    PICTURE X(8).
    02 DET-SEG-LEVEL   PICTURE XX.
    02 DET-STAT-CODE   PICTURE XX.
    02 DET-PROC-OPT    PICTURE XXXX.
    02 FILLER          PICTURE S9(5) COMPUTATIONAL.
    02 DET-SEG-NAME    PICTURE X(8).
    02 DET-LEN-KFB     PICTURE S9(5) COMPUTATIONAL.
    02 DET-NU-SENSESEG PICTURE S9(5) COMPUTATIONAL.
    02 DET-KEY-FB      PICTURE X—X.

```

```

PROCEDURE DIVISION USING IO-PCB, DB-PCB-MAST, DB-PCB-DETAIL
  ENTRY 'DLITCBL'
6
  .
  .
  .
  CALL 'CBLTDLI' USING FUNC-GU, DB-PCB-DETAIL,
    DET-SEG-IN, QUAL-SSA-DET.
7
  .
  CALL 'CBLTDLI' USING COUNT, FUNC-GHU, DB-PCB-MAST,
    MAST-SEG-IN, QUAL-SSA-MAST.
8
  .
  CALL 'CBLTDLI' USING FUNC-GHN, DB-PCB-MAST,
    MAST-SEG-IN.
9
  .
  CALL 'CBLTDLI' USING FUNC-REPL, DB-PCB-MAST,
    MAST-SEG-IN.
10
  .
  GOBACK.
11

```

COBOL LANGUAGE INTERFACE	12
--------------------------	----

Figure 20. Sample COBOL Program (Part 2 of 2)

#### Notes to Figure 20:

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 02-level entries. 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 example in Figure 20 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.
12. IMS supplies a language interface module (DFSLI000). This module must be link-edited 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 link-edits the language interface with the application program. IMSCOBOL is a two-step procedure that compiles and link-edits your program. IMSCOBGO is a three-step procedure that compiles, link-edits, and executes your program in an IMS batch region.

**Related Reading:** For information on how to use these procedures, see *IMS Version 9: Installation Volume 2: System Definition and Tailoring*. If you are using CICS, see *CICS/MVS Installation Guide* for more about installing application programs.

## Coding a CICS Online Program in COBOL

The programs in this section are skeleton online programs in IBM COBOL for z/OS & VM (or VS COBOL II) and OS/VS 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.

CBL APOST

### NOTES

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBLUIB.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 PSB-NAME PIC X(8) VALUE 'CBLPSB '.
77 PCB-FUNCTION PIC X(4) VALUE 'PCB '.
```

```

1      77 TERM-FUNCTION PIC X(4) VALUE 'TERM'.
      77 GHU-FUNCTION PIC X(4) VALUE 'GHU '.
      77 REPL-FUNCTION PIC X(4) VALUE 'REPL'.
      77 SSA1 PIC X(9) VALUE 'AAAA4444 '.
      77 SUCCESS-MESSAGE PIC X(40).
      77 GOOD-STATUS-CODE PIC XX VALUE ' '.
2
      77 GOOD-RETURN-CODE PIC X VALUE LOW-VALUE.
      01 MESSAGE0.
      02 MESSAGE1 PIC X(38).
3
      02 MESSAGE2 PIC XX.
      01 DLI-IO-AREA.
      02 AREA1 PIC X(3).
      02 AREA2 PIC X(37).
LINKAGE SECTION.
COPY DLIUIB.
4,5
      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.
      02 PCB1-STATUS-CODE PIC XX.
      02 PCB1-PROC-OPT PIC XXXX.
6
      02 FILLER PIC S9(5) COMP.
      02 PCB1-SEG-NAME PIC X(8).
      02 PCB1-LEN-KFB PIC S9(5) COMP.
      02 PCB1-NU-SENSEG PIC S9(5) COMP.
      02 PCB1-KEY-FB PIC X(256).
PROCEDURE DIVISION.
* SCHEDULE THE PSB AND ADDRESS THE UIB.
CALL 'CBLTDLI' USING PCB-FUNCTION, PSB-NAME,
7
      ADDRESS OF DLIUIB.
      IF UIBFCTR IS NOT EQUAL LOW-VALUES THEN
* INSERT ERROR DIAGNOSTIC CODE.
EXEC CICS RETURN END-EXEC.
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 GHU-FUNCTION, PCB1,
8
      DLI-IO-AREA, SSA1.
      IF UIBFCTR IS NOT EQUAL GOOD-RETURN-CODE THEN
* INSERT ERROR DIAGNOSTIC CODE
EXEC CICS RETURN END-EXEC.
      IF PCB1-STATUS-CODE IS NOT EQUAL GOOD-STATUS-CODE THEN
* INSERT ERROR DIAGNOSTIC CODE
9
      EXEC CICS RETURN END-EXEC.
* PERFORM SEGMENT UPDATE ACTIVITY
MOVE ..... TO AREA1.
MOVE ..... TO AREA2.
* ISSUE DL/I CALL: REPLACE SEGMENT AT CURRENT POSITION
10
CALL 'CBLTDLI' USING REPL-FUNCTION, PCB1,
      DLI-IO-AREA, SSA1.
      IF UIBFCTR IS NOT EQUAL GOOD-RETURN-CODE THEN
* INSERT ERROR DIAGNOSTIC CODE
EXEC CICS RETURN END-EXEC.

```

```

      IF PCB1-STATUS-CODE IS NOT EQUAL GOOD-STATUS-CODE THEN
*   INSERT ERROR DIAGNOSTIC CODE
      EXEC CICS RETURN END-EXEC.
*   RELEASE THE PSB
      CALL 'CBLTDLI' USING TERM-FUNCTION.
*   OTHER APPLICATION FUNCTION
11,12
      EXEC CICS RETURN END-EXEC.
      GOBACK.

```

**Notes to example:**

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 02-level entries. 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. The linkage section does not contain BLLCELLS with IBM COBOL for z/OS & VM (or VS COBOL II).
5. The COPY DLIUIB statement will be expanded as shown in Figure 27 on page 104.
6. The field UIBPCBAL is redefined as a pointer variable in order to address the special register of IBM COBOL for z/OS & VM (or VS COBOL II). This field contains the address of an area containing the PCB addresses. Do not alter the addresses in the area.
7. 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.
8. The PCB call schedules a PSB for your program to use. The address of the DLIUIB parameter returns the address of DLIUIB.
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 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.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. 'CBLUIB'.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 PSB-NAME PIC X(8) VALUE 'CBLPSB '.
1
77 PCB-FUNCTION PIC X(4) VALUE 'PCB '.
77 TERM-FUNCTION PIC X(4) VALUE 'TERM'.
77 GHU-FUNCTION PIC X(4) VALUE 'GHU '.
77 REPL-FUNCTION PIC X(4) VALUE 'REPL'.
77 SSA1 PIC X(9) VALUE 'AAAA4444 '.
2
77 SUCCESS-MESSAGE PIC X(40).
77 GOOD-STATUS-CODE PIC XX VALUE ' '.
77 GOOD-RETURN-CODE PIC X VALUE LOW-VALUE.
01 MESSAGE.
02 MESSAGE1 PIC X(38).
02 MESSAGE2 PIC XX.
01 DLI-IO-AREA.
3
02 AREA1 PIC X(3).
02 AREA2 PIC X(37).
LINKAGE SECTION.
4
01 BLLCELLS.
02 FILLER PIC S9(8) COMP.
02 UIB-PTR PIC S9(8) COMP.
02 B-PCB-PTRS PIC S9(8) COMP.
02 PCB1-PTR PIC S9(8) COMP.
COPY DLIUIB.
5,6
01 PCB-PTRS.
02 B-PCB1-PTR PIC 9(8) COMP.
01 PCB1.
7
02 PCB1-DBD-NAME PIC X(8).
02 PCB1-SEG-LEVEL PIC XX.
02 PCB1-STATUS-CODE PIC XX.
02 PCB1-PROC-OPT PIC XXXX.
02 FILLER PIC S9(5) COMP.
02 PCB1-SEG-NAME PIC X(8).
02 PCB1-LEN-KFB PIC S9(5) COMP.
02 PCB1-NU-ENSEG PIC S9(5) COMP.
02 PCB1-KEY-FB PIC X(256).
PROCEDURE DIVISION.
8
CALL 'CBLTDLI' USING PCB-FUNCTION, PSB-NAME, UIB-PTR
IF UIBFCTR IS NOT EQUAL LOW-VALUES THEN
    INSERT ERROR DIAGNOSTIC CODE
EXEC CICS RETURN END-EXEC.
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 GHU-FUNCTION, 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.

```

Figure 21. Sample Call-Level OS/V COBOL program (CICS Online) (Part 1 of 2)

```

IF PCB1-STATUS-CODE IS NOT EQUAL GOOD-STATUS-CODE THEN
*   INSERT ERROR DIAGNOSTIC CODE
    EXEC CICS RETURN END-EXEC.
*   PERFORM SEGMENT UPDATE ACTIVITY
MOVE ..... TO AREA1.
MOVE ..... TO AREA2.
*   ISSUE DL/I CALL: REPLACE SEGMENT AT CURRENT POSITION
11  CALL 'CBLTDLI' USING REPL-FUNCTION, PCB1,
    DLI-IO-AREA, SSA1.
IF UIBFCTR IS NOT EQUAL GOOD-RETURN-CODE THEN
*   INSERT ERROR DIAGNOSTIC CODE
    EXEC CICS RETURN END-EXEC.
    IF PCB1-STATUS-CODE IS NOT EQUAL GOOD-STATUS-CODE THEN
*   INSERT ERROR DIAGNOSTIC CODE
    EXEC CICS RETURN END-EXEC.
    RELEASE THE PSB
    CALL 'CBLTDLI' USING TERM-FUNCTION.
12,13 EXEC CICS RETURN END-EXEC.

```

Figure 21. Sample Call-Level OS/V COBOL program (CICS Online) (Part 2 of 2)

#### Notes to Figure 21:

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 as shown in Figure 27 on page 104.
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.

### **Establishing Addressability in a COBOL Program: The Optimization Feature (CICS Online Only)**

If you use the OS/VS COBOL compiler (5740-CB1) with the OPTIMIZE feature, you must use the SERVICE RELOAD compiler control statement in your program to ensure addressability to areas that are defined in the LINKAGE SECTION. If you use the IBM COBOL for z/OS & VM (or VS COBOL II) compiler, the SERVICE RELOAD statement is not required.

The format of the SERVICE RELOAD statement is:

```
SERVICE RELOAD fieldname
```

*fieldname* is the name of a storage area defined in a 01-level statement in the LINKAGE SECTION.

Use the SERVICE RELOAD statement after each statement that modifies addressability to an area in the LINKAGE SECTION. Include the SERVICE RELOAD statement after the label if the statement might cause a branch to another label.

If you specify NOOPTIMIZE when compiling your program, you do not need to use the SERVICE RELOAD statement. However, use this statement to ensure that the program will execute correctly if it is compiled using the OPTIMIZE option.

For more information on using the SERVICE RELOAD statement, see *CICS/ESA Application Programmer's Reference*.

## **Coding a Batch Program in Pascal**

Figure 22 on page 69 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;

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

```

Figure 22. Sample Pascal Program (Part 1 of 2)

```

9      PASTDLI(const
      var  DB_PCB_DETAIL;
      var  DET_SEG_IO_AREA;
      const QUAL_SSA);
10     PASTDLI(const GHU,
      var  DB_PCB_MAST,
      var  MAST_SEG_IO_AREA,
      const QUAL_SSA);
11     PASTDLI(const GHN,
      var  DB_PCB_MAST,
      var  MAST_SEG_IO_AREA);
12     PASTDLI(const REPL,
      var  DB_PCB_MAST,
      var  MAST_SEG_IO_AREA);
13 end;
```

PASCAL LANGUAGE INTERFACE

14

Figure 22. Sample Pascal Program (Part 2 of 2)

#### Notes to Figure 22:

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.
14. You must link-edit your program to the IMS language interface module, DFSLI000, after compiling your program.

## Coding a Batch Program in PL/I

Figure 23 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.

```

/*                                     */
NOTES
/*                                     */
/*                                     */
/*                                     */
DLITPLI: PROCEDURE (IO_PTR_PCB,DB_PTR_MAST,DB_PTR_DETAIL)
1
    OPTIONS (MAIN);
/*                                     */
/*                                     */
/*                                     */
/*                                     */
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);

```

Figure 23. Sample PL/I Program (Part 1 of 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
/*
/*
7  CALL PLITDLI(FOUR, FUNC_GU, DB_PCB_DETAIL,
  DET_SEG_IO_AREA, QUAL_SSA);
.
8  CALL PLITDLI(FOUR, FUNC_GHU, DB_PCB_MAST,
  MAST_SEG_IO_AREA, QUAL_SSA);
.
9  CALL PLITDLI(THREE, FUNC_GHN, DB_PCB_MAST,
  MAST_SEG_IO_AREA);
.
10 CALL PLITDLI(THREE, FUNC_REPL, DB_PCB_MAST,
  MAST_SEG_IO_AREA);
.
11 RETURN
END DLITPLI;

```

PL/I LANGUAGE INTERFACE

12

Figure 23. Sample PL/I Program (Part 2 of 2)

#### Notes to Figure 23:

1. After IMS has loaded the application program's PSB, 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 example in Figure 23 on page 71 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 each SSA's data field 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. The example below 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.
12. IMS provides a language interface module (DFSLI000) which gives a common interface to IMS. This module must be link-edited to the program. If you use the IMS-supplied procedures (IMSPLI or IMSPLIGO), IMS link-edits the language interface module to the application program. IMSPLI is a two-step procedure that compiles and links your program. IMSPLIGO is a three-step procedure that compiles, link-edits, and executes your program in a DL/I batch region. For information on how to use these procedures, see *IMS Version 9: Installation Volume 2: System Definition and Tailoring*.

**Related Reading:** For more information on installing CICS application programs, see *CICS/MVS Installation Guide*.

## Coding a CICS Online Program in PL/I

The program in Figure 24 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.

```

PLIUIB: PROC OPTIONS(MAIN);
  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,
      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,
7
          PSB_NAME,UIBPTR;
      IF UIBFCTR="GOOD_RETURN_CODE THEN DO;
          /* ISSUE DL/I CALL: GET A UNIQUE SEGMENT */
          END;
          CALL PLITDLI(PARM_CT_4,GHU_FUNCTION,PCB1,
8
              DLI_IO_AREA,SSA1;
      IF UIBFCTR="GOOD_RETURN_CODE THEN
9
          IF PCB1_STATUS_CODE=GOOD STATUS CODE THEN
          DO;
              /* PERFORM SEGMENT UPDATE ACTIVITY */
              /* INSERT ERROR DIAGNOSTIC CODE */
              END;
              IF PCB1_STATUS_CODE="GOOD_STATUS_CODE THEN DO;
                  /* INSERT ERROR DIAGNOSTIC CODE */
                  AREA1=.....;
                  AREA2=.....;
                  /* ISSUE DL/I: REPLACE SEGMENT AT CURRENT POSITION */
                  CALL PLITDLI(PARM_CT_4,REPL_FUNCTION,PCB1,
                      DLI_IO_AREA,SSA1);
                  END;
          END;
          IF UIBFCTR="GOOD_RETURN_CODE THEN DO;
              /* ANALYZE UIB PROBLEM */
              .
              .
              /* ISSUE DIAGNOSTIC MESSAGE */
          END;

```

Figure 24. Sample Call-Level PL/I Program (CICS Online) (Part 1 of 2)

```

ELSE IF PCB1_STATUS_CODE=GOOD_STATUS_CODE THEN DO;
  /* EXAMINE PCB1_STATUS_CODE */
  .
  .
  /* ISSUE DIAGNOSTIC MESSAGE */
END;
  /* RELEASE THE PSB */
CALL PLITDLI(PARM_CT_1,TERM_FUNCTION);
11 EXEC CICS RETURN;
12 END PLIUIB;

```

Figure 24. Sample Call-Level PL/I Program (CICS Online) (Part 2 of 2)

#### Notes to Figure 24:

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 SSAs. The SSA character string must reside contiguously in storage. If a call requires two or more SSAs, you may need to define additional areas.
3. The %INCLUDE DLIUIB statement will be expanded as shown in Figure 27 on page 104.
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.



---

## Chapter 3. Defining Application Program Elements

This section describes the elements of your application program that are used with IMS. Your application program must define these elements. This section describes formatting DL/I calls for language interfaces and provides language calls information for assembler language, C language, COBOL, Pascal, and PL/I.

### **In this Chapter:**

- “Formatting DL/I Calls for Language Interfaces”
- “Application Programming for Assembler Language” on page 78
- “Application Programming for C Language” on page 80
- “Application Programming for COBOL” on page 83
- “Application Programming for Pascal” on page 86
- “Application Programming for PL/I” on page 88
- “Relationship of Calls to PCBs” on page 91
- “Specifying the I/O PCB Mask” on page 92
- “Specifying the DB PCB Mask” on page 95
- “Specifying the AIB Mask” on page 98
- “Specifying the AIB Mask for ODBA Applications” on page 99
- “Specifying the UIB (CICS Online Programs Only)” on page 102
- “Specifying the I/O Areas” on page 105
- “Segment Search Arguments” on page 106
- “GSAM Data Areas” on page 111
- “The AIBTDLI Interface” on page 111
- “Specifying the Language Specific Entry Point” on page 112
- “PCB Lists” on page 115
- “The AERTLDI interface” on page 116
- “Language Environment” on page 117
- “Special DL/I Situations” on page 118

**Related Reading:** For detailed information on specific parameters for the DL/I calls, see Chapter 4, “Writing DL/I Calls for Database Management,” on page 121 and Chapter 5, “Writing DL/I Calls for System Services,” on page 149.

---

## Formatting DL/I Calls for Language Interfaces

When you use DL/I calls in a programming language supported by IMS (Assembler, C language, COBOL, Pascal, and 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<sup>®</sup> conforming (CEETDLI)
- A nonspecific language interface for all supported languages (AIBTDLI)
- Language-specific interfaces for all supported languages (xxxTDLI)

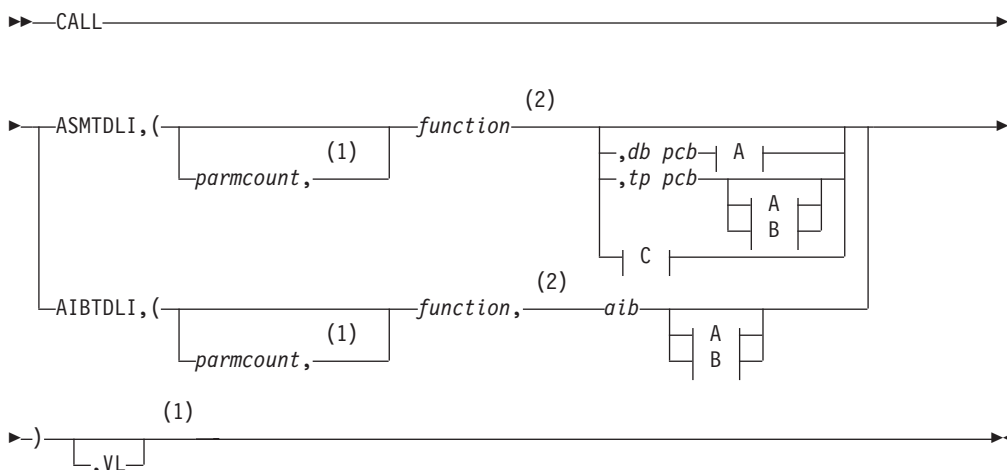
Because each programming language uses a different syntax, the format for calling the language interfaces varies. The following sections describe the detailed format for each supported language.

**Related Reading:** Not every DL/I call uses all the parameters shown. For descriptions of the call functions and the parameters they use, see Chapter 4, "Writing DL/I Calls for Database Management," on page 121 or Chapter 5, "Writing DL/I Calls for System Services," on page 149.

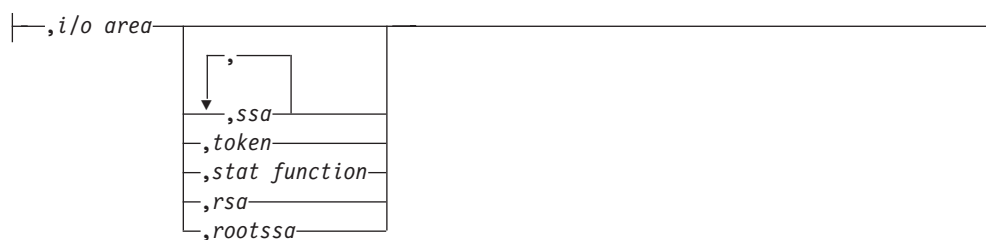
## Application Programming for Assembler Language

This section contains the format, parameters, and DL/I call sample formats for IMS application programs in assembler language. In such 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.

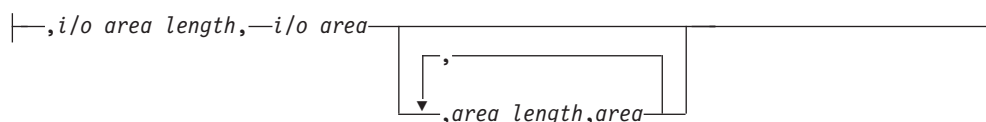
### Format



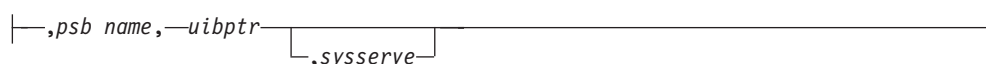
#### A:



#### B:



#### C:



**Notes:**

- 1 Assembler language must use either *parmcount* or **VL**.
- 2 See Chapter 4, “Writing DL/I Calls for Database Management,” on page 121 and Chapter 5, “Writing DL/I Calls for System Services,” on page 149 for descriptions of call functions and parameters.

**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**bb**).

*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. For more information on AIB, see “The AIBTDLI Interface” on page 111.

*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 DL/I Call Formats

**Using the DL/I AIBTDLI interface:**

CALL AIBTDLI,(function,aib,i/o area,ssa1),VL

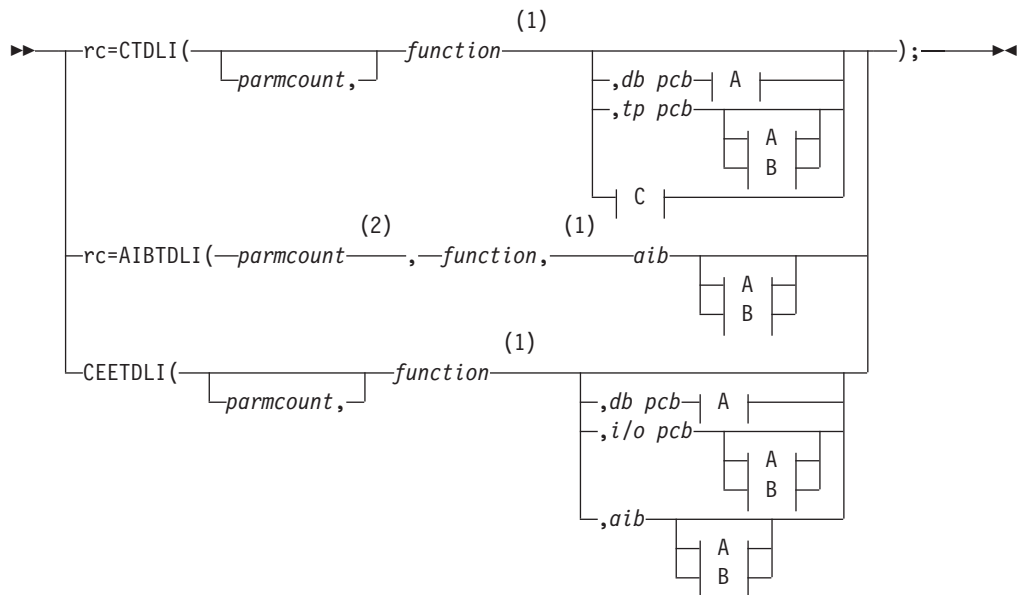
**Using the DL/I language-specific interface:**

CALL ASMTDLI,(function,db pcb,i/o area,ssa1),VL

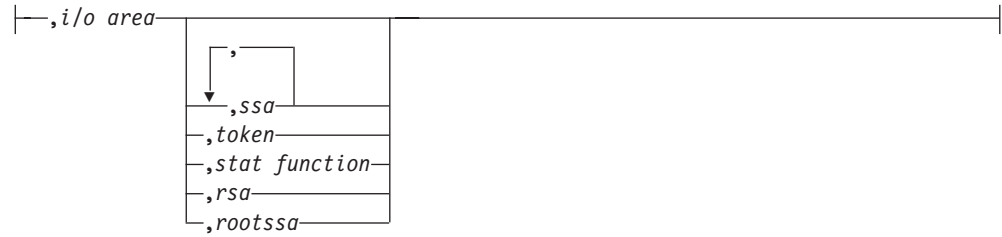
## Application Programming for C Language

This section contains the format, parameters, and DL/I sample call formats for IMS application programs in C language.

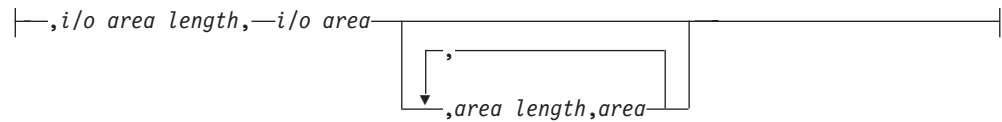
### Format



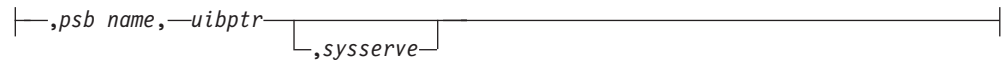
**A:**



**B:**



**C:**



**Notes:**

- 1 See Chapter 4, "Writing DL/I Calls for Database Management," on page 121 and Chapter 5, "Writing DL/I Calls for System Services," on page 149 for descriptions of call functions and parameters.
- 2 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 GUbb).

*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 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. For more information on the AIB, see "The AIBTDLI Interface" on page 111.

*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 DL/I Call Formats

### Using the DL/I CEETDLI interface:

```
#include <leawi.h>
...
CEETDLI (function,db pcb,i/o area,ssal);
```

### Using the DL/I AIBTDLI interface:

```
int rc;
...
rc=AIBTDLI (parmcount,function,aib,i/o area,ssal);
```

### Using the DL/I language-specific interface:

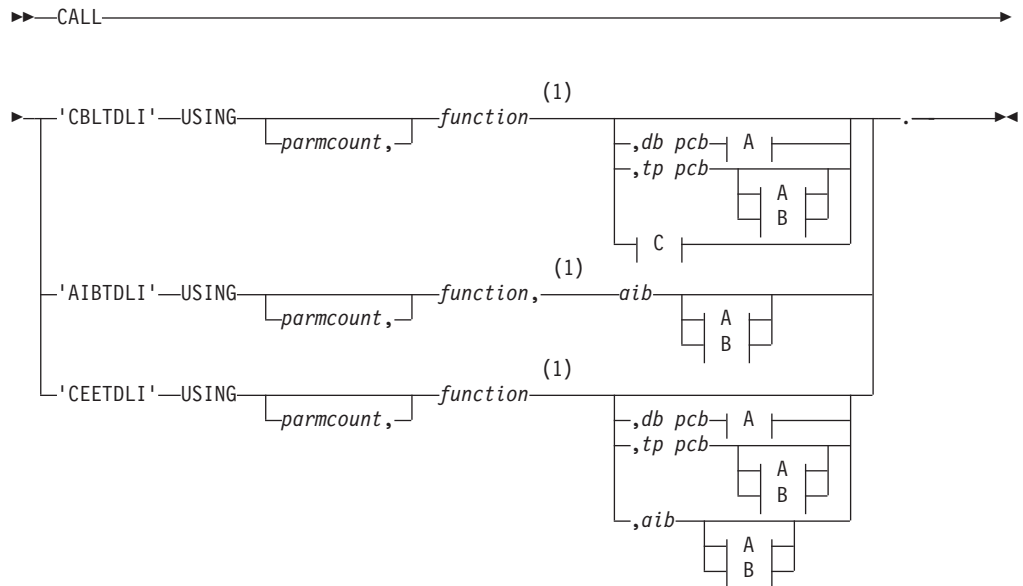
```
#include <ims.h>
int rc;
...
rc=CTDLI (function,db pcb,i/o area,ssal);
```

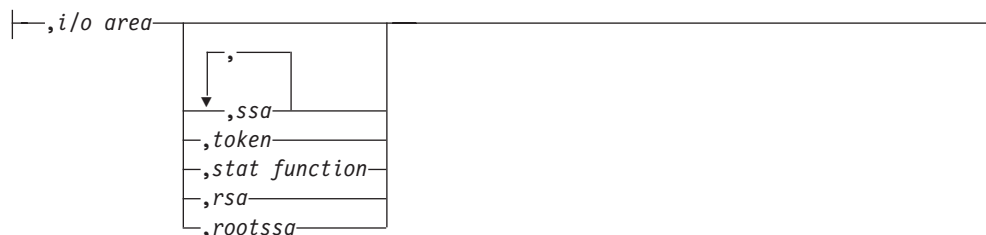
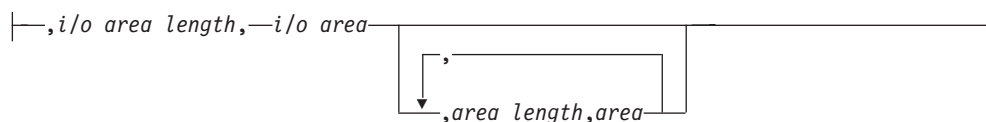
---

## Application Programming for COBOL

This section contains the format, parameters, and DL/I sample call formats for IMS application programs in COBOL.

## Format



**A:****B:****C:****Notes:**

- 1 See Chapter 4, "Writing DL/I Calls for Database Management," on page 121 and Chapter 5, "Writing DL/I Calls for System Services," on page 149 for descriptions of call functions and parameters.

**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 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. For more information on the AIB, see "The AIBTDLI Interface" on page 111.

*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).

*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.

*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 DL/I Call Formats

### 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.
```

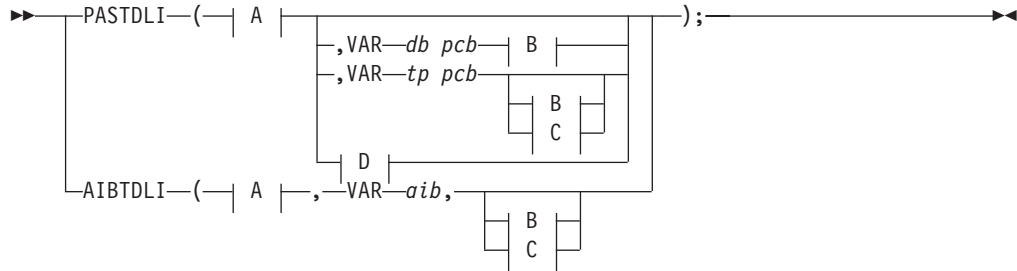
### Using the DL/I language-specific interface:

```
CALL 'CBLTDLI' USING function,db pcb,i/o area,ssa1.
```

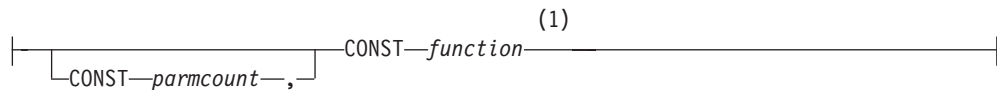
## Application Programming for Pascal

This section contains the format, parameters, and DL/I sample call formats for IMS application programs in Pascal.

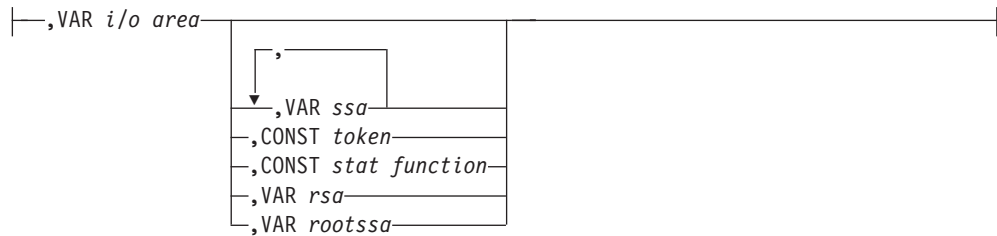
### Format



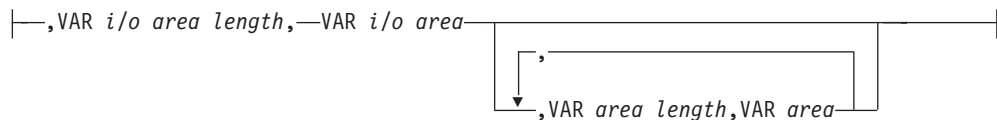
**A:**



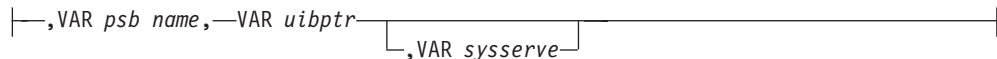
**B:**



**C:**



**D:**



**Notes:**

- 1 See Chapter 4, "Writing DL/I Calls for Database Management," on page 121 and Chapter 5, "Writing DL/I Calls for System Services," on page 149 for descriptions of call functions and parameters.

## 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 GUbb).

### *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. For more information on the AIB, see "The AIBTDLI Interface" on page 111.

### *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 DL/I Call Formats

### Using the DL/I AIBTDLI interface:

```
AIBTDLI(CONST function,
VAR aib,
VAR i/o area,
VAR ssal);
```

### Using the DL/I language-specific interface:

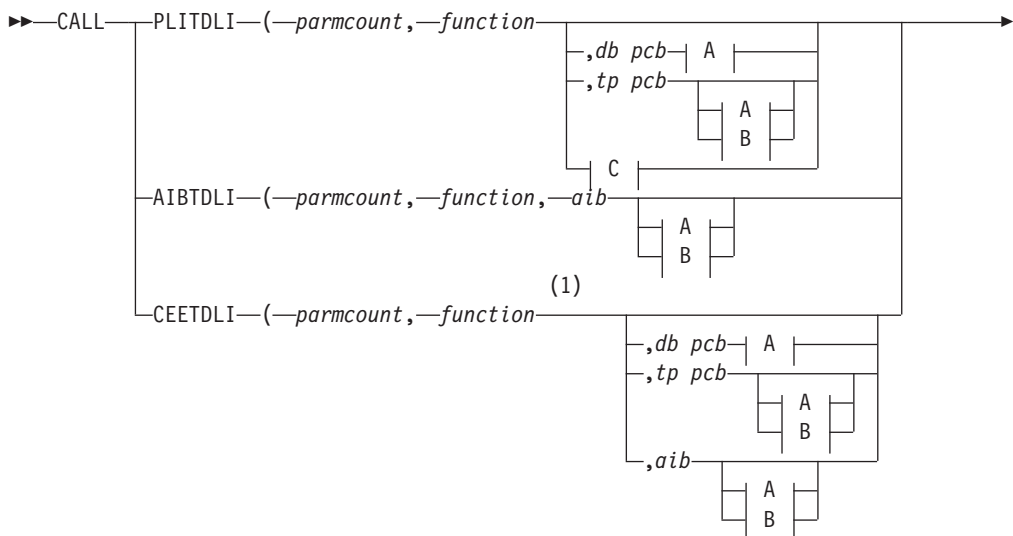
```
PASTDLI(CONST function,
VAR db pcb,
VAR i/o area,
VAR ssal);
```

## Application Programming for PL/I

This section contains the format, parameters, and DL/I sample call formats for IMS application programs in PL/I.

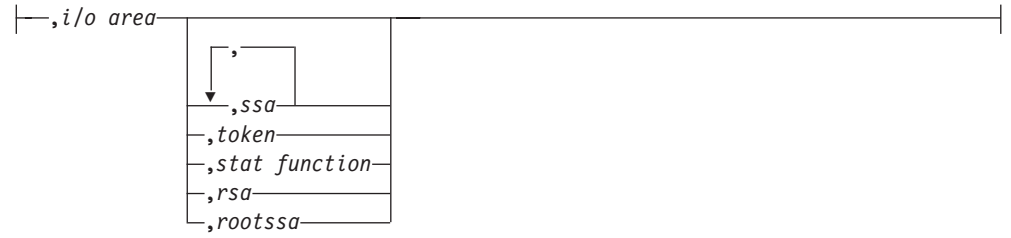
**Exception:** For the PLITDLI interface, all parameters except *parmcount* are indirect pointers; for the AIBTDLI interface, all parameters are direct pointers.

## Format

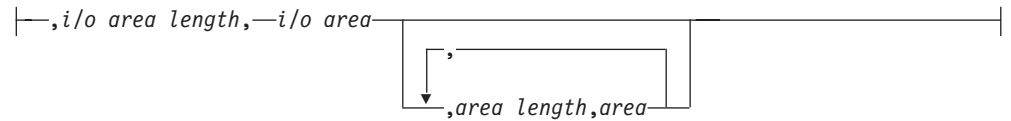


▶-);

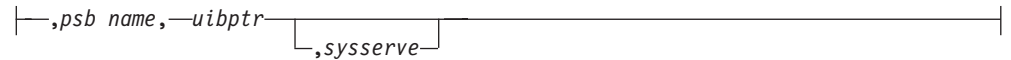
**A:**



**B:**



**C:**



**Notes:**

- 1 See Chapter 4, "Writing DL/I Calls for Database Management," on page 121 and Chapter 5, "Writing DL/I Calls for System Services," on page 149 for descriptions of call functions and parameters.

## 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 fixed-character (4-byte) variable left-justified, blank padded character string containing the call function to be used (such as GUbb).

*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. For more information on the AIB, see "The AIBTDLI Interface" on page 111.

*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 DL/I Call Formats

### 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);
```

## Relationship of Calls to PCBs

Table 15 shows the relationship of calls to full function (FF), main storage database (MSDB), data entry database (DEDB), I/O, and general sequential access method (GSAM) PCBs.

Table 15. Call Relationship to PCBs

CALL	FF PCBs	MSDB PCBs	DEDB PCBs	I/O PCBs	GSAM PCBs
CHKP				X	
CLSE					X
DEQ			X	X	
DLET	X	X	X		
FLD		X	X		
GHN	X	X	X		
GHNP	X	X	X		
GHU	X	X	X		
GN	X	X	X	X	X
GNP	X	X	X		
GSCD <sup>1</sup>	X	X	X	X	
GU	X	X	X	X	X
INIT				X	
INQY	X	X	X	X	X
ISRT	X	X	X	X	X
LOG				X	
OPEN					X
PCB <sup>2</sup>					
POS			X		
REPL	X	X	X		
ROLB				X	
ROLL <sup>2</sup>					
ROLS				X	
SETS/SETU				X	
SNAP <sup>3</sup>	X	X	X	X	
STAT <sup>3</sup>	X				
SYNC				X	
TERM <sup>2</sup>					
XRST				X	

**Note:**

1. GSCD is a Product-sensitive programming interface.
2. The PCB, ROLL, and TERM calls do not have an associated PCB.
3. SNAP is a Product-sensitive programming interface.
4. STAT is a Product-sensitive programming interface.

## 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.

Table 16 shows the fields that the I/O PCB contains, their lengths, and the applicable environment for each field.

Table 16. I/O PCB Mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Logical terminal name <sup>1</sup>	8	X		X		
Reserved for IMS <sup>2</sup>	2	X		X		
Status code <sup>3</sup>	2	X	X	X	X	X
4-Byte Local date and time <sup>4</sup>						
Date	2	X		X		
Time	2	X		X		
Input message sequence number <sup>5</sup>	4	X		X		
Message output descriptor name <sup>6</sup>	8	X		X		
Userid <sup>7</sup>	8	X		X		
Group name <sup>8</sup>	8	X		X		
<b>12-Byte Time Stamp <sup>9</sup></b>						
Date	4	X		X		
Time	6	X		X		
UTC Offset	2	X		X		
Userid Indicator <sup>10</sup>	1	X		X		
Reserved for IMS <sup>2</sup>	3					

### Notes:

#### 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 hhmmss. 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 9: Administration Guide: Transaction Manager*.

### 7. Userid

The use of this field is connected with RACF<sup>®</sup> 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 9: Administration Guide: Transaction Manager*.

## 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:

<b>Date</b>	yyyydddf
	This packed-decimal date contains the year (yyyy), day of the year (ddd), and a valid packed-decimal + sign such as (f).
<b>Time</b>	hhmmssthmi ju
	This packed-decimal time consists of hours, minutes, and seconds (hhmms) and fractions of the second to the microsecond (thmi ju). <b>No</b> packed-decimal sign is affixed to this part of the time stamp.
<b>UTC Offset</b>	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 page 93 always contains the local date and time.

**Related Reading:** For a more detailed description of the internal packed-decimal time-format, see *IMS Version 9: DBRC Guide and Reference*.

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 Table 17. (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.

**Related Reading:** For more information about GSAM DB PCB Masks, see "GSAM DB PCB Masks" on page 111.

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 17. DB PCB Mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Database name <sup>1</sup>	8	X	X		X	

Table 17. DB PCB Mask (continued)

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Segment level number <sup>2</sup> on page 96	2	X	X		X	
Status code <sup>3</sup>	2	X	X		X	
Processing options <sup>4</sup>	4	X	X		X	
Reserved for IMS <sup>5</sup>	4	X	X		X	
Segment name <sup>6</sup>	8	X	X		X	
Length of key feedback area <sup>7</sup>	4	X	X		X	
Number of sensitive segments <sup>8</sup>	4	X	X		X	
Key feedback area <sup>9</sup>	var length	X	X		X	

**Notes:****1. Database Name**

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 "INIT Call" on page 159.

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 9: Messages and Codes, Volume 1* 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 Length of Key Feedback Area field (as described earlier in these notes). 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 the length of the key feedback area described above 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.

## Specifying the AIB Mask

The 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 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 Table 18. The table's notes describe the contents of each field.

Table 18. AIB Fields

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
AIB identifier <sup>1</sup>	8	X	X	X	X	X
DFSAIB allocated length <sup>2</sup>	4	X	X	X	X	X
Subfunction code <sup>3</sup>	8	X	X	X	X	X
Resource name <sup>4</sup>	8	X	X	X	X	X
Reserved <sup>5</sup>	16					
Maximum output area length <sup>6</sup>	4	X	X	X	X	X
Output area length used <sup>7</sup>	4	X	X	X	X	X
Reserved <sup>8</sup>	12					
Return code <sup>9</sup>	4	X	X	X	X	X
Reason code <sup>10</sup>	4	X	X	X	X	X
Error code extension <sup>11</sup>	4	X		X		
Resource address <sup>12</sup>	4	X	X	X	X	X
Reserved <sup>13</sup>	48					

### Notes:

#### 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**

This 16-byte field is reserved.

6. **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.

7. **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.

8. **Reserved**

This 12-byte field is reserved.

9. **Return code (AIBRETRN)**

When the call is completed, this 4-byte field contains the return code.

10. **Reason Code (AIBREASN)**

When the call is completed, this 4-byte field contains the reason code.

11. **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.

12. **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.

13. **Reserved**

This 48-byte field is reserved.

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.

**Related Reading:** For more information about the return and reason codes, see *IMS Version 9: Messages and Codes, Volume 1*.

---

## Specifying the AIB Mask for ODBA Applications

Table 19 describes the fields for specifying the AIB mask for ODBA applications. The notes below the table describe the contents of each field.

*Table 19. AIB Fields for ODBA Applications' Use*

Description	Byte Length	DB/DC	IMS DB	DCCTL	DB Batch	TM Batch
AIB identifier <sup>1</sup>	8	X	X	X	X	X
DFSAIB allocated length <sup>2</sup>	4	X	X	X	X	X



Table 19. AIB Fields for ODBA Applications' Use (continued)

Description	Byte Length	DB/DC	IMS DB	DCCTL	DB Batch	TM Batch
Subfunction code <sup>3</sup>	8	X	X	X	X	X
Resource name #1 <sup>4</sup>	8	X	X	X	X	X
Resource name #2 <sup>5</sup>	8					
Reserved <sup>6</sup>	8	X				
Maximum output area length <sup>7</sup>	4	X	X	X	X	X
Output area length used <sup>8</sup>	4	X	X	X	X	X
Reserved <sup>9</sup>	12					
Return code <sup>10</sup>	4	X	X	X	X	X
Reason code <sup>11</sup>	4	X	X	X	X	X
Error code extension <sup>12</sup>	4	X				
Resource address #1 <sup>13</sup>	4	X	X	X	X	X
Resource address #2 <sup>14</sup>	4					
Resource address #3 <sup>15</sup>	4					
Reserved <sup>16</sup>	40					
Reserved for ODBA <sup>17</sup>	136					

**Notes:****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 264 bytes for ODBA. 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) #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.

**5. Resource Name (AIBRSNM2) #2**



Specify a 4-character ID of ODBA startup table DFSxxxx0, where xxxx is a four-character ID.

6. **Reserved**

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**

This 12-byte field is reserved.

10. **Return code (AIBRETRN)**

When the call is completed, this 4-byte field contains the return code.

11. **Reason Code (AIBREASN)**

When the call is completed, this 4-byte field contains the reason code.

12. **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.

13. **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.

14. **Resource Address (AIBRSA2) #2**

This 4-byte field is reserved for ODBA.

15. **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.

16. **Reserved**

This 40-byte field is reserved.

17. **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.

## AIB Examples

### COBOL AIB Example

```

01  AIB
   02  AIBRID          PIC x(8).
   02  AIBRLN         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 AIBRSVAVE        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 Example

```

DFS AIB  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 (Figure 25 on page 103 for VS COBOL II, or Figure 26 for OS/VS COBOL). "Coding a CICS Online Program in COBOL" on page 63 shows how to establish addressability to the UIB. Figure 27 on page 104 shows the fields defined when you use the COBOL COPY DLIUIB statement.
- In PL/I programs, use a %INCLUDE DLIUIB statement (Figure 28 on page 105). "Coding a CICS Online Program in PL/I" on page 74 shows how to establish addressability to the UIB.

- In assembler language programs, use the DLIUIB macro (Figure 29 on page 105). “Coding a CICS Online Program in Assembler Language” on page 53 shows how to establish addressability to the UIB.

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 Chapter 17, “CICS-DL/I User Interface Block Return Codes,” on page 303.

Immediately after the statement that defines the UIB in your program, you must define the PCB address list and the PCB mask.

Figure 25 provides an example of using the COPY DLIUIB statement in a VS COBOL II program:

```
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.
    .
    .
    .
```

*Figure 25. Defining the UIB, PCB Address List, and the PCB Mask for VS COBOL II*

Figure 26 provides an example of using the COPY DLIUIB statement in an OS/VS COBOL program.

```

LINKAGE SECTION.
01 BLL CELLS.
  02 FILLER          PIC S9(8) COMP.
  02 UIB-PTR        PIC S9(8) COMP.
  02 PCB-LIST-PTR   PIC S9(8) COMP.
  02 PCB1-PTR       PIC S9(8) COMP.

COPY DLIUIB.

01 PCB-ADDRESS-LIST.
  02 PCB1-LIST-PTR PIC S9(8) COMP.
01 PCB1.
  02 PCB1-DBD-NAME PIC X(8).
  02 PCB1-SEG-LEVEL PIC XX.
  .
  .
  .

```

Figure 26. Defining the UIB, PCB Address List, and the PCB Mask for OS/VS COBOL

Figure 27 provides an example of using the COBOL COPY DLIUIB statement.

```

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 ' '.

```

Figure 27. The COBOL DLIUIB Copy Book

The values placed in level 88 entries are not printable. They are described in Chapter 17, "CICS-DL/I User Interface Block Return Codes," on page 303. The meanings of the field names and their hexadecimal values are shown below:

**FCNORESP**

Normal response X'00'

**FCNOTOPEN**

Not open X'0C'

**FCINVREQ**

Invalid request X'08'

**FCINVPCB**

Invalid PCB X'10'

- DLPSBNF**  
PSB not found X'01'
- DLTASKNA**  
Task not authorized X'02'
- DLPSBSCH**  
PSB already scheduled X'03'
- DLLANGCON**  
Language conflict X'04'
- DLPSBFAIL**  
PSB initialization failed X'05'
- DLPSBNA**  
PSB not authorized X'06'
- DLTERMNS**  
Termination not successful X'07'
- DLFUNCNS**  
Function unscheduled X'08'
- DLINA**  
DL/I not active X'FF'

Figure 28 shows you how to define the UIB, PCB address list, and 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  */
    
```

Figure 28. Defining the UIB, PCB Address List, and the PCB Mask for PL/I

Figure 29 shows you how to define the UIB, PCB address list, and 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

Figure 29. Defining the UIB, PCB Address List, and the PCB Mask for Assembler Language

---

## Specifying the I/O Areas

Use an I/O area to pass segments between your program and IMS. What the I/O area contains depends on the type of call you are issuing:

- When you retrieve a segment, IMS DB 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 DB 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 adds to the database. 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.

---

## Segment Search Arguments

This section describes the coding rules and provides coding formats and examples for defining SSAs in Assembler language, C language, COBOL, Pascal, and PL/I.

### SSA Coding Rules

The rules for coding an SSA are as follows:

- 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. Table 20 on page 107 lists the relational operators.

Table 20. Relational Operators

Symbolic	Alphabetic	Meaning
=b or b=	EQ	Equal to
>= or =>	GE	Greater than or equal to
<= or =<	LE	Less than or equal to
>b or b>	GT	Greater than
<b or 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.
- 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.

## SSA Coding Restrictions

The 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 9: Utilities Reference: Database and Transaction Manager*.

## SSA Coding Formats

This section shows examples of coding formats for assembler language, C language, COBOL, Pascal, and PL/I.

### Assembler Language SSA Definition Examples

Figure 30 on page 108 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')'

```

Figure 30. Example Code: \* CONSTANT AREA

This SSA looks like this:

```
ROOTbbbb(KEYbbbbbb=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,
        "%-8.8s(%-8.8s%2.2s%-6.6s)",
        "ROOT", "KEY", "=", "vvvvv");

```



Alternatively, if only the value were changing, the **sprintf** call can be:

```
sprintf(qual_ssa,
        "ROOT (KEY =%-6.6s)", "vvvvv");
/* 12345678 12345678 */
```

In both cases, the SSA looks like this:

```
ROOTbbbb(KEYbbbbbb=vv...v)
```

These SSAs are both taken from the C skeleton program shown in Figure 19 on page 57. To see how the SSAs are used in DL/I calls, refer to that program.

### COBOL SSA Definition Examples

An unqualified SSA without 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:

```
ROOTbbbb(KEYbbbbbb=vv...v)
```

These SSAs are both taken from the COBOL skeleton program in Figure 20 on page 61. To see how they are used in a DL/I call, refer to that program.

### Pascal SSA Definition Examples

An unqualified SSA without command codes looks like this in Pascal:

```
type
  STRUCT = record
    SEG_NAME : ALFA;
    BLANK    : CHAR;
  end;
const
  UNQUAL_SSA = STRUCT('NAME', '');
```

You can, of course, declare this as a single string:

```
const
  UNQUAL_SSA = 'NAME  ';
```

The SSA shown in Figure 31 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.

```
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', ')');
```

Figure 31. Qualified SSA without Command Codes

This SSA looks like this:

```
ROOTbbbb(KEYbbbbbb=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 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.

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), you should not need to change the SSAs 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:

```
ROOTbbbb(KEYbbbbbb=vv...v)
```

Both of these SSAs are taken from the PL/I skeleton program shown in Figure 23 on page 71. To see how they are used in DL/I calls, refer to that program.

## GSAM Data Areas

This section shows how to code GSAM data areas. GSAM applies only to batch and BMPs. The PCB mask and the RSA that you use in a GSAM call have special formats.

### GSAM DB PCB Masks

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.

**Related Reading:** For more information on GSAM, see Chapter 10, “Processing GSAM Databases,” on page 219.

### GSAM RSAs

The RSA (record search argument) is an 8-byte token that can be returned on GN and ISRT calls. The application program can save the RSA for use in a subsequent GU call.

**Related Reading:** For more information on RSAs for GSAM, see Chapter 10, “Processing GSAM Databases,” on page 219.

## The AIBTDLI Interface

This section explains how to use the application interface block (AIB), an 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.

### Overview

When you use the AIBTDLI interface, you 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 AIBTDLI 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 AIBTDLI 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. For a

generated program specification block (GPSB), the I/O PCB is generated with the PCB name IOPCBbbb, and the modifiable alternate PCB is generated with the PCB name TPPCB1b.

Because you can pass the PCB name, 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.

**Related Reading:** For more information about PSBGEN, see *IMS Version 9: Utilities Reference: System*.

## Defining Storage for the AIB

The AIB resides in user-defined storage that is passed to IMS for DL/I calls that use the AIBTDLI interface. When the call is completed, the AIB is updated by IMS.

**Recommendation:** Allocate at least 128 bytes of storage for the AIB.

---

## Specifying the Language Specific Entry Point

IMS gives control to an application program through an *entry point*. The formats for coding entry statements in Assembler language, C language, COBOL, Pascal, and PL/I are shown in these sections: “Assembler Language,” “C Language” on page 113, “COBOL” on page 113, “Pascal” on page 114, and “PL/I” on page 114. Your entry point must refer to the PCBs in the order in which they have been defined in the PSB.

IMS passes the PCB pointers to a PL/I program differently than it passes them to assembler language, C language, COBOL, or Pascal programs. 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 that is specified during PSBGEN is consistent with the language of the program.

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 “PCB Call (CICS Online Programs Only)” on page 171.

Application interfaces that use the AIB structure (AIBTDLI or CEETDLI) use the PCB name rather than the PCB structure, and they 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). For more information, see “Specifying the UIB (CICS Online Programs Only)” on page 102.

## 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 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

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

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

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 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.

## Interface Considerations

This section explains the interfaces: CEETDLI and AIBTDLI, and AERTDLI.

### CEETDLI

The 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 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>.

### AIBTDLI

The considerations are:

- When using the AIBTDLI 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 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.

### AERTDLI

The considerations 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.

---

## PCB Lists

This section describes the formats of PCB lists and GPSB PCB lists, and provides a description of PCBs in various types of application programs.

### Format of a PCB List

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 DCCTL. GSAM PCBs can be used with DCCTL.

### 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. (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 IOPCBbb. The name of the alternate PCB is TPPCB1bb.

## PCB Summary

This section summarizes the information concerning I/O PCBs and alternate PCBs in various types of application programs. You should read this section if you intend to issue system service requests.



**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.

Table 21 summarizes the I/O PCB and alternate PCB information.

*Table 21. 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 <sup>1</sup>	No	Yes
DB Batch <sup>2</sup>	Yes	Yes
TM Batch <sup>3</sup>	Yes	Yes
CICS DBCTL <sup>4</sup>	No	No
CICS DBCTL <sup>5</sup>	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 AERTLDI interface

This section explains how to use the AIB with ODBA applications.

### Overview

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.

## Defining Storage for the AIB

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.

**Requirement:** Allocate 264 bytes of storage for the AIB.

---

## Language Environment

IBM Language Environment for MVS and VM 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 application programs, 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/C++ for MVS/ESA™
- IBM COBOL for MVS and VM
- IBM PL/I for MVS and VM

These programs can be produced by programs coded in Assembler. All these programs can use CEETDLI, the Language Environment-provided language-independent interface to IMS, as well as older language-dependent interfaces to IMS, such as CTDLI, CBLTDLI, and PLITDLI.

Although they do not conform to Language Environment, programs that are compiled with the following older compilers can run under Language Environment:

- IBM C/370™
- IBM VS COBOL II
- IBM OS PL/I

These programs cannot use CEETDLI, but they can use the older language-dependent interfaces to IMS.

**Related Reading:** For more information about Language Environment, see *IBM Language Environment for MVS and VM Programming Guide*.

## 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 that Language Environment provides. 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 *IBM Language Environment for MVS and VM Programming Guide* and *Language Environment for MVS & VM Installation and Customization*.

## LANG= Option on PSBGEN for PL/I Compatibility with Language Environment

For IMS PL/I applications running in a compatibility mode that uses the PLICALLA entry point, you must specify LANG=PLI on the PSBGEN, or you can 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. Table 22 summarizes when you can use LANG=b and LANG=PLI.

Table 22. Using LANG= Option in a Language Environment for PL/I Compatibility

Compile EXEC statement is PARM=(...,SYSTEM(IMS)...	and entry point name is PLICALLA	Then LANG= is as stated below:
Yes	Yes	LANG=PLI
Yes	No	LANG=b or LANG=PLI
No	No	Not valid for IMS PL/I applications
No	Yes	LANG=PLI

**Restriction:** PLICALLA is only valid for PL/I compatibility with Language Environment. If a PL/I application program using PLICALLA entry at link-edit time is link-edited using Language Environment with the PLICALLA entry, the link-edit will work; however, you must use LANG=PLI. If the application program is re-compiled using PL/I for z/OS & VM Version 1 Release 1, and then link-edited using Language Environment Version 1 Release 2 or later, the link-edit will fail. You must remove the PLICALLA entry statement from the link-edit.

## Special DL/I Situations

This section contains information on:

- Application programs scheduled against HALDBs
- Mixed-language programming using the extended addressing capabilities of MVS/ESA
- Preloaded programs using COBOL compiler options

## 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.

**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.

### Initial Load of HALDBs

If you load a new HALDB that contains logical relationships, the logical child segments are not loaded as part of the load step. Add logical children through normal update processing after the database is loaded.

When a program accesses a partition for the first time, an indicator records that the PSB accessed the partition. Commands can operate against a partition currently not in use. A DFS05651 message results if a BMP uses a partition and the command was against that partition. If an application attempts to access data from a stopped partition, a pseudo abend results or the application receives a BA status code. If the partition starts before the application attempts to access data in that partition again, the DL/I call succeeds.

## 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.

## 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 routine retention, see *IBM Language Environment for MVS & VM Programming Guide* and *IBM Language Environment for MVS & VM Installation and Customization*.

## Extended Addressing Capabilities of MVS/ESA

The two modes in MVS/ESA 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.

## 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.

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 4. Writing DL/I Calls for Database Management

This chapter describes the calls you can use with IMS DB to perform database management functions in your application program. Calls within this section are in alphabetical order.

Each call description contains:

- A syntax diagram
- Definitions for parameters that are available to the call
- Details on how to use the call in your application program
- Restrictions on call usage, where applicable

Each parameter is described as an input parameter or output parameter. “Input” refers to input to IMS from the application program. “Output” refers to output from IMS to the application program.

Database management calls must use either *db pcb* or *aib* parameters. The syntax diagrams for these calls begin with the *function* parameter. The call, call interface (*xxxTDLI*), and *parmcount* (if it is required) are not included in the syntax diagrams.

### **In this Chapter:**

- “CIMS Call”
- “CLSE Call” on page 123
- “DEQ Call” on page 123
- “DLET Call” on page 125
- “FLD Call” on page 126
- “GN/GHN Call” on page 128
- “GNP/GHNP Call” on page 132
- “GU/GHU Call” on page 135
- “ISRT Call” on page 138
- “OPEN Call” on page 141
- “POS Call” on page 142
- “REPL Call” on page 145

**Related Reading:** For specific information about coding your program in assembler language, C language, COBOL, Pascal, and PL/I, see Chapter 3, “Defining Application Program Elements,” on page 77. For information on the DL/I calls used for transaction management and EXEC DLI commands used in CICS, see *IMS Version 9: Application Programming: Transaction Manager* and *IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS*.

---

### CIMS Call

The CIMS call is used to initialize and terminate the ODBA interface in an z/OS application region.

### Format

▶▶—CIMS—*aib*—◀◀

Call Name	DB/DC	IMS DB	DCCTL	DB Batch	TM Batch
CIMS	X	X			

## Parameters

*aib*

Specifies the application interface block (AIB) that is used for the call. This parameter is an input and output parameter.

The following fields must be initialized in the AIB:

### AIBID

Eye-catcher. This 8-byte field must contain DFSAIBbb.

### AIBLEN

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

### AIBRSNM1

Character value. This field is optional.

### AIBSFUNC

Subfunction code. This field must contain one of the 8-byte subfunction codes as follows:

#### INIT

AIBRSNM2. A four-character ID of the ODBA startup table (optional).

#### TERM

AIBRSNM2. A four-character ID of the ODBA startup table representing the IMS connection that is to be terminated.

#### TALL

Terminate all IMS connections.

## Usage

The CIMS call is used by an application program that is running in an application address space to establish/terminate the ODBA environment.

### INIT**bbbb**

The CIMS subfunction INIT must be issued by the application to establish the ODBA environment in the z/OS application address space.

Optionally, AIBRSNM2 can specify the 4-character ID of the ODBA Startup table member. This is the member named DFSxxxx0 where xxxx is equal to the 4-character ID. If AIBRSNM2 is specified, ODBA will attempt to establish a connection to the IMS specified in the DFSxxxx0 member after the ODBA environment has been initialized in the z/OS application address space.

### TERM**bbbb**

The CIMS subfunction TERM can be issued to terminate one and only one IMS connection. AIBRSNM2 specifies the 4-character ID of the startup table member representing the IMS connection to be terminated. Upon completion of the TERM subfunction the ODBA environment will remain intact in the z/OS application address space.

**Note:** If the application that issued CIMS INIT chooses to return to the operating system following completion of the CIMS TERM, the address space will experience a system abend A03. This can be avoided by issuing the CIMS TALL prior to returning to the operating system

**TALL***bbbb*

The CIMS subfunction TALL must be issued to terminate all IMS connections and terminate the ODBA environment in the application address space.

**CLSE Call**

The close (CLSE) call is used to explicitly close a GSAM database. For more information on GSAM, see Chapter 10, "Processing GSAM Databases," on page 219.

**Format**



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For GSAM:	CLSE	X	X	X	X	X

**Parameters**

*gsam pcb*

Specifies the GSAM PCB for the call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB length. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a GSAM PCB.

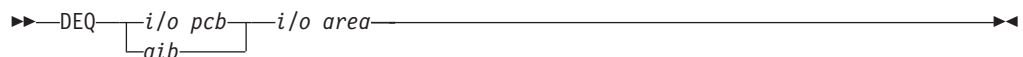
**Usage**

For information on using CLSE, see "Explicitly Opening and Closing a GSAM Database" on page 222.

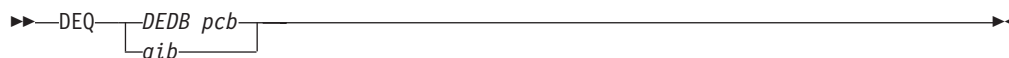
**DEQ Call**

The Dequeue (DEQ) call is used to release a segment that is retrieved using the Q command code.

**Format (Full Function)**



## Format (Fast Path DEDB)



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function and DEDB:	DEQ	X	X		X	

## Parameters

### *DEQB pcb* (Fast Path only)

Specifies any DEDB PCB for the call.

### *i/o pcb* (full function only)

Specifies the I/O PCB for the DEQ call. This is an input and output parameter.

### *aib*

Specifies the AIB for the call. This is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name IOPCBbbb.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area* (full function only)

Specifies the 1-byte area containing a letter (A-J), which represents the lock class of the locks to be released. This is a mandatory input parameter.

## Usage

The DEQ call releases all segments that are retrieved using the Q command code, except:

- Segments modified by your program, until your program reaches a commit point
- Segments required to keep your position in the hierarchy, until your program moves to another database record
- A class of segments that has been locked using a different lock class

If your program only reads segments, it can release them by issuing a DEQ call. If your program does not issue a DEQ call, IMS releases the reserved segments when your program reaches a commit point. By releasing the segments with a DEQ call before your program reaches a commit point, you make them available to other programs more quickly.

For more information on the relationship between the DEQ call and the Q command code, see “Reserving Segments for the Exclusive Use of Your Program” on page 256.



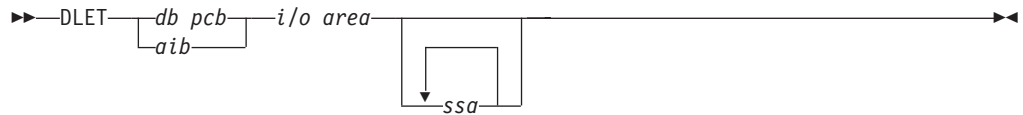
## Restrictions

In a CICS DL/I environment, calls made from one CICS (DBCTL) system are supported in a remote CICS DL/I environment, if the remote environment is also CICS (DBCTL).

## DLET Call

The Delete (DLET) call is used to remove a segment and its dependents from the database.

## Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function:	DLET	X	X		X	
For DEDB:	DLET	X	X			
For MSDB:	DLET	X				

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the I/O area in your program that communicates with IMS. This parameter is an input parameter. Before deleting a segment, you must first issue a Get Hold call to place the segment in the I/O area. You can then issue the DLET call to delete the segment and its dependents in the database.

### *ssa*

Specifies the SSAs, if any, to be used in the call. This parameter is an input parameter. The SSAs you supply in the call point to data areas in your program

in which you have defined the SSAs for the call. You can use only one SSA in the parameter. This parameter is optional for the DLET call.

## Usage

The DLET call must be preceded by one of the three Get Hold calls. When you issue the DLET call, IMS deletes the held segment, along with all its physical dependents from the database, *regardless of whether your program is sensitive to all of these segments*. IMS rejects the DLET call if the preceding call for the PCB was not a Get Hold, REPL, or DLET call. If the DLET call is successful, the previously retrieved segment and all of its dependents are removed from the database and cannot be retrieved again.

If the Get Hold call that precedes the DLET call is a path call, and you do not want to delete all the retrieved segments, you must indicate to IMS which of the retrieved segments (and its dependents, if any) you want deleted; to do this, specify an unqualified SSA for that segment. Deleting a segment this way automatically deletes all dependents of the segment. Only one SSA is allowed in the DLET call, and this is the only time an SSA is applicable in a DLET call.

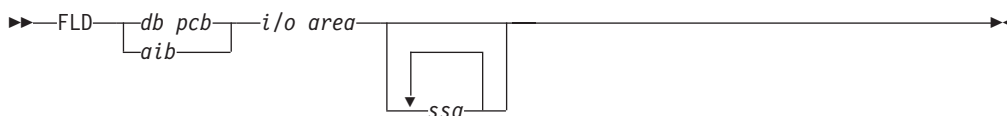
No command codes apply to the DLET call. If you use a command code in a DLET call, IMS disregards the command code.

---

## FLD Call

The Field (FLD) call is used to access a field within a segment for MSDBs or DEDBs.

## Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For MSDB:	FLD	X				
For DEDB:	FLD	X	X			

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies your program's I/O area, which contains the field search argument (FSA) for this call. This parameter is an input parameter.

*ssa*

Specifies the SSAs, if any, that you want to use in this call. You can use up to 15 SSAs in this input parameter. The SSAs that you supply will point to those data areas that you have defined for the call. This parameter is optional for the FLD call.

## Usage

Use the FLD call to access and change the contents of a field within a segment.

The FLD call does two things for you: it compares the value of a field to the value you supply (FLD/VERIFY), and it changes the value of the field in the way that you specify (FLD/CHANGE).

All DL/I command codes are available to DEDBs, using the FLD call. The FLD call formats for DEDBs are the same as for other DL/I calls. So, if your MSDBs have been converted to DEDBs, you do not need to change application programs that use the FLD call. For more information on the FLD call, see "Updating Segments in an MSDB or DEDB: REPL, DLET, ISRT, and FLD" on page 231.

You can also use the FLD call in application programs for DEDBs, instead of the combination of GHU, REPL, and DL/I calls.

## FSAs

The field search argument (FSA) is equivalent to the I/O area that is used by other DL/I database calls. For a FLD call, data is not moved into the I/O area; rather, the FSAs are moved into the I/O area.

Multiple FSAs are allowed on one FLD call. This is specified in the FSA's connector field. Each FSA can operate on either the same or different fields within the target segment.

The FSA that you reference in a FLD call contains five fields. The rules for coding these fields are as follows:

**Field name**

This field must be 8 bytes long. If the field name you are using is less than 8 bytes, the name must be left-justified and padded on the right with blanks.

**FSA status code**

This field is 1 byte. IMS returns one of the following status codes to this area after a FLD call:

- b** Successful
- A** Invalid operation
- B** Operand length invalid

- C** Invalid call—program tried to change key field
- D** Verify check was unsuccessful
- E** Packed decimal or hexadecimal field is invalid
- F** Program tried to change an unowned segment
- G** Arithmetic overflow
- H** Field not found in segment

**Op code**

This 1-byte field contains one of the following operators for a change operation:

- +** To add the operand to the field value
- To subtract the operand from the field value
- =** To set the field value to the value of the operand

For a verify operation, this field must contain one of the following:

- E** Verify that the field value and the operand are equal.
- G** Verify that the field value is greater than the operand.
- H** Verify that the field value is greater than or equal to the operand.
- L** Verify that the field value is less than the operand.
- M** Verify that the field value is less than or equal to the operand.
- N** Verify that the field value is not equal to the operand.

**Operand**

This variable length field contains the value that you want to test the field value against. The data in this field must be the same type as the data in the segment field. (You define this in the DBD.) If the data is hexadecimal, the value in the operand is twice as long as the field in the database. If the data is packed decimal, the operand does not contain leading zeros, so the operand length might be shorter than the actual field. For other types of data, the lengths must be equal.

**Connector**

This 1-byte field must contain a blank if this is the last or only FSA, or an asterisk (\*) if another FSA follows this one.

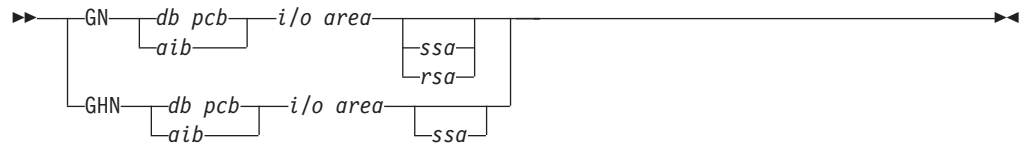
The format of SSAs in FLD calls is the same as the format of SSAs in DL/I calls. If no SSA exists, the first segment in the MSDB or DEDB is retrieved.

For more information on the FLD call and some examples, see “Processing MSDBs and DEDBs” on page 231.

**GN/GHN Call**

The Get Next (GN) call is used to retrieve segments sequentially from the database. The Get Hold Next (GHN) is the hold form for a GN call.

**Format**



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function:	GN/GHN	X	X		X	
For GSAM:	GN	X	X	X	X	X
For DEDB:	GN	X	X	X		
For MSDB:	GN	X				

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the I/O area. This parameter is an output parameter. When you issue one of the Get calls successfully, IMS returns the requested segment to this area. If your program issues any path calls, the I/O area must be long enough to hold the longest path of concatenated segments following a path call. This area always contains left-justified segment data. The I/O area points to the first byte of this area.

When you use the GN call with GSAM, the area named by the *i/o area* parameter contains the record you are retrieving.

### *ssa*

Specifies the SSAs, if any, to be used in the call. This parameter is an input parameter. The SSAs you supply in the call point to data areas in your program in which you have defined the SSAs for the call. You can use up to 15 SSAs in the parameter. This parameter is optional for the GN call.

### *rsa*

Specifies the area in your program where the RSA for the record should be returned. This output parameter is used for GSAM only and is optional. See "GSAM RSAs" on page 111 for more information on RSAs.

## Usage, Get Next (GN)

A Get Next (GN) call is a request for a segment, as described by the SSAs you supply, that is linked to the call that was issued prior to the GN call. IMS starts its search at the current position.

When you use the GN call:

- Processing moves forward from current position (unless the call includes the F command code).
- IMS uses the current position (that was set by the previous call) as the search starting point.
- The segment retrieved is determined by a combination of the next sequential position in the hierarchy and the SSAs included in the call.
- Be careful when you use GN, because it is possible to use SSAs that force IMS to search to the end of the database without retrieving a segment. This is particularly true with the “not equal” or “greater than” relational operators.

A GN call retrieves the next segment in the hierarchy that satisfies the SSAs that you supplied. Because the segment retrieved by a GN call depends on the current position in the hierarchy, GN is often issued after a GU call. If no position has been established in the hierarchy, GN retrieves the first segment in the database. A GN call retrieves a segment or path of segments by moving forward from the current position in the database. As processing continues, IMS looks for segments at each level to satisfy the call.

**Example:** Sequential retrieval in a hierarchy is always top to bottom and left to right. For example, if you repeatedly issue unqualified GN calls against the hierarchy in Figure 32, IMS returns the segment occurrences in the database record in this order:

1. A1 (the root segment)
2. B1 and its dependents (C1,D1,F1,D2,D3,E1,E2, and G1)
3. H1 and its dependents (I1,I2,J1, and K1).

If you issue an unqualified GN again, after IMS has returned K1, IMS returns the root segment occurrence whose key follows segment A1 in the database.

A GN call that is qualified with the segment type can retrieve all the occurrences of a particular segment type in the database.

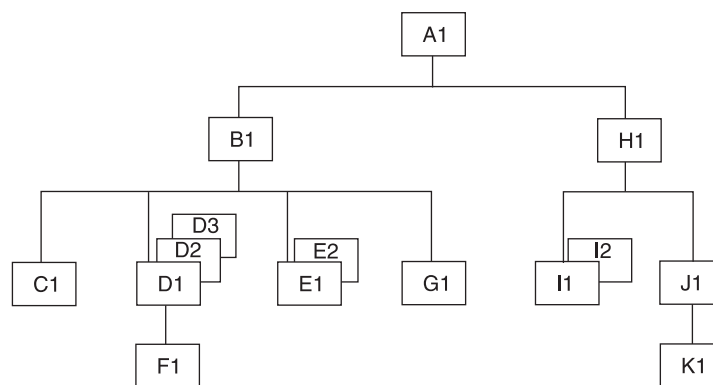


Figure 32. Hierarchic Sequence

**Example:** If you issue a GN call with qualified SSAs for segments A1 and B1, and an unqualified SSA for segment type D, IMS returns segment D1 the first time you issue the call, segment D2 the second time you issue the call, and segment D3 the third time you issue the call. If you issue the call a fourth time, IMS returns a status code of GE, which means that IMS could not find the segment you requested.

You can use unqualified GN calls to retrieve all of the occurrences of a segment in a hierarchy, in their hierarchic sequence, starting at the current position. Each unqualified GN call retrieves the next sequential segment forward from the current position. For example, to answer the processing request:

Print out the entire medical database.

You would issue an unqualified GN call repeatedly until IMS returned a GB status code, indicating that it had reached the end of the database without being able to satisfy your call. If you issued the GN again after the GB status code, IMS would return the first segment occurrence in the database.

Like GU, a GN call can have as many SSAs as the hierarchy has levels. Using fully qualified SSAs with GN calls clearly identifies the hierarchic path and the segment you want, thus making it useful in documenting the call.

A GN call with an unqualified SSA retrieves the next occurrence of that segment type by going forward from the current position.

GN with a qualified SSA retrieves the next occurrence of the specified segment type that satisfies the SSA.

When you specify a GN that has multiple SSAs, the presence or absence of unqualified SSAs in the call has no effect on the operation unless you use command codes on the unqualified SSAs. IMS uses only qualified SSAs plus the last SSA to determine the path and retrieve the segment. Unspecified or unqualified SSAs for higher-level segments in the hierarchy mean that any high-level segment that is the parent of the correct lower-level, specified or qualified segment will satisfy the call.

A GN call with an SSA that is qualified on the key of the root can produce different results from a GU with the same SSA, depending on the position in the database and the sequence of keys in the database. If the current position in the database is beyond a segment that would satisfy the SSA, the segment is not retrieved by the GN. GN returns the GE status code if both the following conditions are met:

- The value of the key in the SSA has an upper limit that is set, for example, to less-than-or-equal-to the value.
- A segment with a key greater than the value in the SSA is found in a sequential search before the specified segment is found.

GN returns the GE status code, even though the specified segment exists and would be retrieved by a GU call.

## Usage, Get Hold Next (GHN)

Before your program can delete or replace a segment, it must retrieve the segment and indicate to IMS that it is going to change the segment in some way. The program does this by issuing a Get call with a “hold” before deleting or replacing

the segment. When the program has successfully retrieved the segment with a Get Hold call, it can delete the segment or change one or more fields (except the key field) in the segment.

The only difference between Get calls with a hold and Get calls without a hold is that the hold calls can be followed by REPL or DLET.

The hold status on the retrieved segment is canceled and must be reestablished before you reissue the DLET or REPL call. After issuing a Get Hold call, you can issue more than one REPL or DLET call to the segment if you do not issue intervening calls to the same PCB.

After issuing a Get Hold call, if you find out that you do not need to update it after all, you can continue with other processing without releasing the segment. The segment is freed as soon as the current position changes—when you issue another call to the same PCB that you used for the Get Hold call. In other words, a Get Hold call must precede a REPL or DLET call. However, issuing a Get Hold call does not require you to replace or delete the segment.

## Usage, HDAM, PHDAM, or DEDB Database with GN

For database organizations other than HDAM, PHDAM, and DEDB, processing the database sequentially using GN calls returns the root segments in ascending key sequence. However, the order of the root segments for a HDAM, PHDAM, or DEDB database depends on the randomizing routine that is specified for that database. Unless a sequential randomizing routine was specified, the order of the root segments in the database is not in ascending key sequence.

For a hierarchic direct access method (HDAM, PHDAM) or a DEDB database, a series of unqualified GN calls or GN calls that are qualified only on the root segment:

1. Returns all the roots from one anchor point
2. Moves to the next anchor point
3. Returns the roots from the anchor point

Unless a sequential randomizing routine was specified, the roots on successive anchor points are not in ascending key sequence. One situation to consider for HDAM, PHDAM, and DEDB organizations is when a GN call is qualified on the key field of the root segment with an equal-to operator or an equal-to-or-greater-than operator. If IMS has an existing position in the database, it checks to ensure that the requested key is equal to or greater than the key of the current root. If it is not, a GE status code is returned. If it is equal to or greater than the current key and is not satisfied using the current position, IMS calls the randomizing routine to determine the anchor point for that key. IMS tries to satisfy the call starting with the first root of the selected anchor.

## Restriction

You can use GN to retrieve the next record of a GSAM database, but GHN is not valid for GSAM.

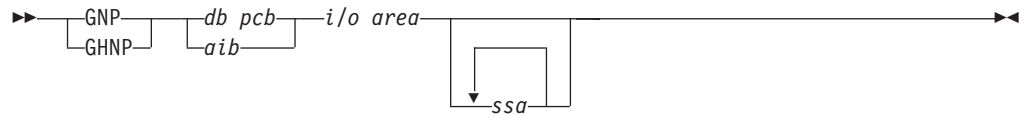
---

## GNP/GHNP Call

The Get Next in Parent (GNP) call is used to retrieve dependents sequentially. The Get Hold Next in Parent (GHNP) call is the hold form for the GNP call.



## Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function:	GNP/GHNP	X	X		X	
For DEDB:	GNP/GHNP	X	X	X		
For MSDB:	GNP/GHNP	X				

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the I/O area. This parameter is an output parameter. When you issue the Get call successfully, IMS returns the requested segment to this area. If your program issues any path calls, the I/O area must be long enough to hold the longest path of concatenated segments following a path call. The segment data that this area contains is always left-justified. The I/O area points to the first byte of this area.

### *ssa*

Specifies the SSAs, if any, to be used in the call. This parameter is an input parameter. The SSAs you supply in the call point to data areas in your program in which you have defined the SSAs for the call. You can use up to 15 SSAs for this parameter. This parameter is optional for the GNP call.

## Usage, Get Next in Parent (GNP)

A GNP call retrieves segments sequentially. The difference between a GN and a GNP is that GNP limits the segments that can satisfy the call to the dependent segments of the established parent.

An unqualified GNP retrieves the first dependent segment occurrence under the current parent. If your current position is already on a dependent of the current parent, an unqualified GNP retrieves the next segment occurrence.

If you are moving forward in the database, even if you are not retrieving every segment in the database, you can use GNP to restrict the returned segments to only those children of a specific segment.

### Linking with Previous DL/I Calls

A GNP call is linked to the previous DL/I calls that were issued by your program in two ways:

- Current position: The search for the requested segment starts at the current position established by the preceding GU, GN, or GNP call.
- Parentage: The search for the requested segment is limited to the dependents of the lowest-level segment most recently accessed by a GU or GN call. Parentage determines the end of the search and is in effect only following a successful GU or GN call.

### Processing with Parentage

You can set parentage in two ways:

- By issuing a successful GU or GN call. When you issue a successful GU or GN call, IMS sets parentage at the lowest-level segment returned by the call. Issuing another GU or GN call (but against a different PCB) does not affect the parentage that you set using the first PCB in the previous call. An unsuccessful GU or GN call cancels parentage.
- By using the P command code with a GU, GN, or GNP call, you can set parentage at any level.

### How DL/I Calls Affect Parentage

A GNP call does not affect parentage unless it includes the P command code.

Unless you are using a secondary index, REPL does not affect parentage. If you are using a secondary index, and you replace the indexed segment, parentage is lost. For more information, see “How Secondary Indexing Affects Your Program” on page 211.

A DLET call does not affect parentage unless you delete the established parent. If you do delete the established parent, you must reset parentage before issuing a GNP call.

ISRT affects parentage only when you insert a segment that is not a dependent of the established parent. In this case, ISRT cancels parentage. If the segment you are inserting is a dependent at some level of the established parent, parentage is unaffected. For example, in Figure 38 on page 194, assume segment B11 is the established parent. Neither of these two ISRT calls would affect parentage:

```
ISRT  ABBBBBBB(AKEYBBBB=bA1)
      BBBBBBBB(BKEYBBBB=bB11)
      CBBBBBBB
```

```
ISRT  ABBBBBBB(AKEYBBBB=bA1)
      BBBBBBBB(BKEYBBBB=bB11)
      CBBBBBBB(CKEYBBBB=bC111)
      DBBBBBBB
```

The following ISRT call **would** cancel parentage, because the F segment is not a direct dependent of B, the established parent:

```
ISRT  Abbbbbbb(AKEYbbbb=bA1)
      Fbbbbbbbbb
```

You can include one or more SSAs in a GNP call. The SSAs can be qualified or unqualified. Without SSAs, a GNP call retrieves the next sequential dependent of the established parent. The advantage of using SSAs with GNP is that they allow you to point IMS to a specific dependent or dependent type of the established parent.

A GNP with an unqualified SSA sequentially retrieves the dependent segment occurrences of the segment type you have specified under the established parent.

A GNP with a qualified SSA describes to IMS the segment you want retrieved or the segment that is to become part of the hierarchic path to the segment you want retrieved. A qualified GNP describes a unique segment only if it is qualified on a unique key field and not a data field or a non unique key field.

A GNP with multiple SSAs defines the hierarchic path to the segment you want. If you specify SSAs for segments at levels above the established parent level, those SSAs must be satisfied by the current position at that level. If they cannot be satisfied using the current position, a GE status code is returned and the existing position remains unchanged. The last SSA must be for a segment that is below the established parent level. If it is not, a GP status code is returned. Multiple unqualified SSAs establish the first occurrence of the specified segment type as part of the path you want. If some SSAs between the parent and the requested segment in a GNP call are missing, they are generated internally as unqualified SSAs. This means that IMS includes the first occurrence of the segment from the missing SSA as part of the hierarchic path to the segment you have requested.

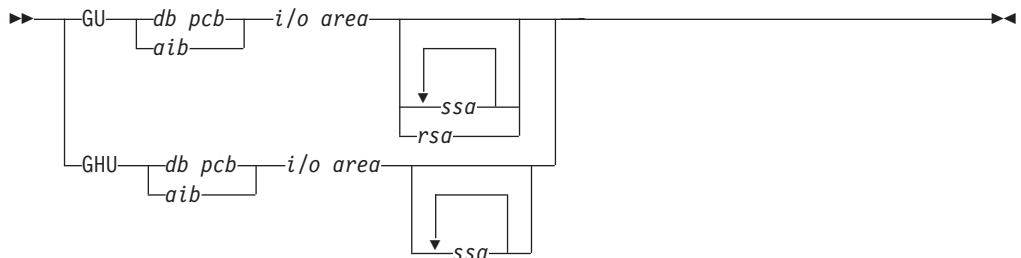
### Usage, Get Hold Next in Parent (GHNP)

Retrieval for the GHNP call is the same as for the GHN call. For more information, see "Usage, Get Hold Next (GHN)" on page 131.

## GU/GHU Call

The Get Unique (GU) call is used to directly retrieve segments and to establish a starting position in the database for sequential processing. The Get Hold Unique (GHU) is the hold form for a GU call.

### Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function:	GU/GHU	X	X		X	
For GSAM:	GU	X	X	X	X	X

	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For DEDB:	GU	X	X	X		
For MSDB:	GU	X				

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the I/O area. This parameter is an output parameter. When you issue one of the Get calls successfully, IMS returns the requested segment to this area. If your program issues any path calls, the I/O area must be long enough to hold the longest path of concatenated segments following a path call. The segment data that this area contains is always left-justified. The I/O area points to the first byte of this area.

When you use the GU call with GSAM, the area named by the *i/o area* parameter contains the record you are retrieving.

### *ssa*

Specifies the SSAs, if any, to be used in the call. This parameter is an input parameter. The SSAs you supply in the call point to data areas in your program in which you have defined the SSAs for the call. You can use up to 15 SSAs for the parameter. This parameter is optional for the GU call.

### *rsa*

Specifies the area in your program that contains the record search argument. This required input parameter is only used for GSAM. See "GSAM RSAs" on page 111 for more information on RSAs.

## Usage, Get Unique (GU)

GU is a request for a segment, as described by the SSAs you supply. You use it when you want a specific segment. You can also use it to establish your position in the database.

The GU call is the only call that can establish position backward in the database. (The GN and GNP calls, when used with the F command code, can back up in the

database, but with limitations. “The F Command Code” on page 31 explains this.) Unlike GN and GNP, a GU call does not move forward in the database automatically.

If you issue the same GU call repeatedly, IMS retrieves the same segment each time you issue the call. If you want to retrieve only particular segments, use fully qualified GUs for these segments instead of GNs. If you want to retrieve a specific segment occurrence or obtain a specific position within the database, use GU.

If you want to retrieve a specific segment or to set your position in the database to a specific place, you generally use qualified GU calls. A GU call can have the same number of SSAs as the hierarchy has levels, as defined by the DB PCB. If the segment you want is on the fourth level of the hierarchy, you can use four SSAs to retrieve the segment. (No reason would ever exist to use more SSAs than levels in the hierarchy. If your hierarchy has only three levels, you would never need to locate a segment lower than the third level.) The following is additional information for using the GU call with the SSA:

- A GU call with an unqualified SSA at the root level attempts to satisfy the call by starting at the beginning of the database. If the SSA at the root level is the only SSA, IMS retrieves the first segment in the database.
- A GU call with a qualified SSA can retrieve the segment described in the SSA, regardless of that segment’s location relative to current position.
- When you issue a GU that mixes qualified and unqualified SSAs at each level, IMS retrieves the first occurrence of the segment type that satisfies the call.
- If you leave out an SSA for one of the levels in a GU call that has multiple SSAs, IMS assumes an SSA for that level. The SSA that IMS assumes depends on current position:
  - If IMS has a position established at the missing level, the SSA that IMS uses is derived from that position, as reflected in the DB PCB.
  - If IMS does not have a position established at the missing level, IMS assumes an unqualified SSA for that level.
  - If IMS moves forward from a position established at a higher level, IMS assumes an unqualified SSA for that level.
  - If the SSA for the root level is missing, and IMS has position established on a root, IMS does not move from that root when trying to satisfy the call.

## Usage, Get Hold Unique (GHU)

Before your program can delete or replace a segment, it must retrieve the segment and indicate to IMS that it is going to change the segment in some way. The program does this by issuing a Get call with a “hold” before deleting or replacing the segment. Once the program has successfully retrieved the segment with a Get Hold call, it can delete the segment or change one or more fields (except the key field) in the segment.

The only difference between Get calls with a hold and without a hold is that the hold calls can be followed by a REPL or DLET call.

The hold status on the retrieved segment is canceled and must be reestablished before you reissue the DLET or REPL call. After issuing a Get Hold call, you can issue more than one REPL or DLET call to the segment if you do not issue intervening calls to the same PCB.

After issuing a Get Hold call, if you find out that you do not need to update it after all, you can continue with other processing without releasing the segment. The

segment is freed as soon as the current position changes—when you issue another call to the same PCB you used for the Get Hold call. In other words, a Get Hold call must precede a REPL or DLET call. However, issuing a Get Hold call does not require you to replace or delete the segment.

## Restriction

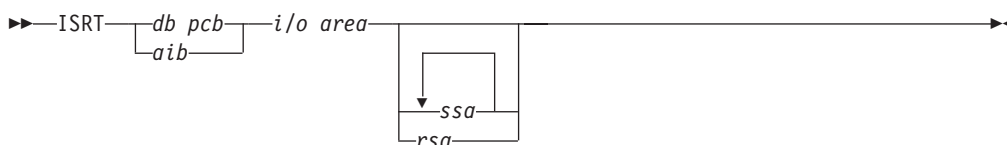
You can use GU to retrieve the record with the RSA you provide with a GSAM database, but GHU is not valid for GSAM.

---

## ISRT Call

The Insert (ISRT) call is used to load a database and to add one or more segments to the database. You can use ISRT to add a record to the end of a GSAM database or for an alternate PCB that is set up for IAFP processing.

## Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function:	ISRT	X	X		X	
For GSAM:	ISRT	X	X	X	X	X
For DEDB:	ISRT	X	X	X		
For MSDB:	ISRT	X				

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the I/O area. This parameter is an input parameter. When you want to add a new segment to the database, you place the new segment in this area before issuing the ISRT call. This area must be long enough to hold the longest segment that IMS returns to this area. For example, if none of the segments your program retrieves or updates is longer than 48 bytes, your I/O area should be 48 bytes.

If your program issues any path calls, the I/O area must be long enough to hold the longest concatenated segment following a path call. The segment data that this area contains is always left-justified. The I/O area points to the first byte of this area.

When you use the ISRT call with GSAM, the area named by the *i/o area* parameter contains the record you want to add. The area must be long enough to hold these records.

*ssa*

Specifies the SSAs, if any, to be used in the call. This parameter is an input parameter. The SSAs you supply in the call point to data areas in your program in which you have defined the SSAs for the call. You can use up to 15 SSAs on the call. This parameter is required.

*rsa*

Specifies the area in your program where the RSA should be returned by DL/I. This output parameter is used for GSAM only and is optional. See “GSAM RSAs” on page 111 for more information on RSAs.

## Usage

Your program uses the ISRT call to initially load a database and to add information to an existing one. The call looks the same in either case. However, the way it is used is determined by the processing option in the PCB. This section explains how you use ISRT to add segments to an existing database.

ISRT can add new occurrences of an existing segment type to a HIDAM, PHIDAM, HISAM, HDAM, PHDAM, DEDB, or MSDB database.

**Restriction:** New segments cannot be added to a HSAM database unless you reprocess the whole database or add the new segments to the end of the database.

Before you issue the ISRT call, build the new segment in the I/O area. The new segment fields must be in the same order and of the same length as defined for the segment. (If field sensitivity is used, they must be in the order defined for the application program’s view of the segment.) The DBD defines the fields that a segment contains and the order in which they appear in the segment.

### Root Segment Occurrence

If you are adding a root segment occurrence, IMS places it in the correct sequence in the database by using the key you supply in the I/O area. If the segment you are inserting is not a root, but you have just inserted its parent, you can insert the child segment by issuing an ISRT call with an unqualified SSA. You must build the new segment in your I/O area before you issue the ISRT call. Also, you use an unqualified SSA when you insert a root. When you are adding new segment occurrences to an existing database, the segment type must have been defined in the DBD. You can add new segment occurrences directly or sequentially after you have built them in the program’s I/O area. At least one SSA is required in an ISRT



call; the last (or only) SSA specifies the segment being inserted. To insert a path of segments, you can set the D command code for the highest-level segment in the path.

### Insert Rules

If the segment type you are inserting has a unique key field, the place where IMS adds the new segment occurrence depends on the value of its key field. If the segment does not have a key field, or if the key is not unique, you can control where the new segment occurrence is added by specifying either the FIRST, LAST, or HERE insert rule. Specify the rules on the RULES parameter of the SEGM statement of DBDGEN for this database.

**Related Reading:** For information on performing a DBDGEN, see *IMS Version 9: Utilities Reference: Database and Transaction Manager*.

The rules on the RULES parameter are as follows:

<b>FIRST</b>	IMS inserts the new segment occurrence before the first existing occurrence of this segment type. If this segment has a nonunique key, IMS inserts the new occurrence before all existing occurrences of that segment that have the same key field.
<b>LAST</b>	IMS inserts the new occurrence after the last existing occurrence of the segment type. If the segment occurrence has a nonunique key, IMS inserts the new occurrence after all existing occurrences of that segment type that have the same key.
<b>HERE</b>	IMS assumes you have a position on the segment type from a previous IMS call. IMS places the new occurrence before the segment occurrence that was retrieved or deleted by the last call, which is immediately before current position. If current position is not within the occurrences of the segment type being inserted, IMS adds the new occurrence before all existing occurrences of that segment type. If the segment has a nonunique key and the current position is not within the occurrences of the segment type with equal key value, IMS adds the new occurrence before all existing occurrences that have equal key fields.

You can override the insert rule of FIRST with the L command code. You can override the insert rule of HERE with either the F or L command code. This is true for HDAM and PHDAM root segments and for dependent segments in any type of database that have either nonunique keys or no keys at all.

An ISRT call must have at least one unqualified SSA for each segment that is added to the database. Unless the ISRT is a path call, the lowest-level SSA specifies the segment being inserted. This SSA must be unqualified. If you use the D command code, all the SSAs below and including the SSA containing the D command code must be unqualified.

Provide qualified SSAs for higher levels to establish the position of the segment being inserted. Qualified and unqualified SSAs can be used to specify the path to the segment, but the last SSA must be unqualified. This final SSA names the segment type to be inserted.

If you supply only one unqualified SSA for the new segment occurrence, you must be sure that current position is at the correct place in the database to insert that segment.



### Mix Qualified and Unqualified SSAs

You can mix qualified and unqualified SSAs, but the last SSA must be unqualified. If the SSAs are unqualified, IMS satisfies each unqualified SSA with the first occurrence of the segment type, assuming that the path is correct. If you leave out an SSA for one of the levels in an ISRT with multiple SSAs, IMS assumes an SSA for that level. The SSA that IMS assumes depends on current position:

- If IMS has a position established at the missing level, the SSA that IMS uses is derived from that position, as reflected in the DB PCB.
- If IMS does not have a position established at the missing level, IMS assumes an unqualified SSA for that level.
- If IMS moves forward from a position established at a higher level, IMS assumes an unqualified SSA for that level.
- If the SSA for the root level is missing, and IMS has position established on a root, IMS does not move from that root when trying to satisfy the call.

### Using SSAs with the ISRT Call

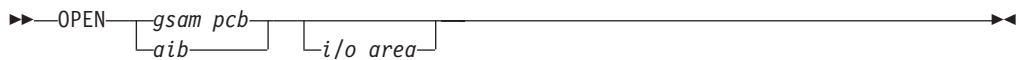
Using SSAs with ISRT is a good way to check for the parent segments of the segment you want to insert. You cannot add a segment unless its parent segments exist in the database. Instead of issuing Get calls for the parents, you can define a fully qualified set of SSAs for all the parents and issue the ISRT call for the new segment. If IMS returns a GE status code, at least one of the parents does not exist. You can then check the segment level number in the DB PCB to find out which parent is missing. If the level number in the DB PCB is 00, IMS did not find any of the segments you specified. A 01 means that IMS found only the root segment; a 02 means that the lowest-level segment that IMS found was at the second level; and so on.

---

## OPEN Call

The OPEN call is used to explicitly open a GSAM database.

### Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For GSAM:	OPEN	X	X	X	X	X

### Parameters

*gsam pcb*

Specifies the GSAM PCB for the call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name of a GSAM PCB.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the kind of data set you are opening. This parameter is an input parameter.

## Usage

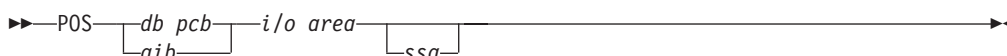
For more information, see “Explicitly Opening and Closing a GSAM Database” on page 222.

## POS Call

A qualified Position (POS) call is used to retrieve the location of a specific sequential dependent segment. In addition to location, a qualified POS call using an SSA for a committed segment will return the sequential dependent segment (SDEP) time stamp and the ID of the IMS owner that inserted it. For more information about the qualified POS call, see “Locating the Last Inserted Sequential Dependent Segment” on page 243.

An unqualified POS points to the logical end of the sequential dependent segment (SDEP) data. By default, an unqualified POS call returns the DMACNXTS value, which is the next SDEP CI to be allocated. Because this CI has not been allocated, its specification without the EXCLUDE keyword will often result in a DFS2664A message from the SDEP utilities.

## Format



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For DEDB:	POS	X	X			

## Parameters

*db pcb*

Specifies the DB PCB for the DEDB that you are using for this call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the DEDB that you are using for this call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBb̄b̄.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the I/O area in your program that you want to contain the positioning information that is returned by a successful POS call. This parameter is an output parameter. The I/O area should be long enough to contain all entries returned. IMS returns an entry for each area in the DEDB.

The I/O area on a POS call contains six words with nine potential fields of data for each entry returned. Each word is four bytes and each field is eight bytes. When the successful POS is an unqualified call, the I/O area contains the length (LL), followed by as many entries as existing areas within the database. You provide one of the five keywords in position one (word 0 word 1) of the I/O area, which determines what data is returned in the I/O area.

Table 23 lists the five keywords and the data that an unqualified POS call returns based on the keyword you choose for position one.

*Table 23. Unqualified POS Call: Keywords and Map of the I/O Area Returned*

Keyword	word 0	word 1	word 2	word 3	word 4	word 5
<null>		Field 1		Field 2	Field 4	Field 5
V5SEGRBA		Field 1		Field 3		<null>
PCSEGRTS		Field 1		Field 3		Field 6
PSSEGHWM		Field 1		Field 3		Field 7
PCHSEGTS		Field 1		Field 8		Field 6
PCLBTSGTS		Field 1		Field 9		Field 6

**Field 1**

Area name

**Field 2**

Sequential dependent next to allocate CI

**Field 3**

Local sequential dependent next segment

**Field 4**

Unused CIs in sequential dependent part

**Field 5**

Unused CIs in independent overflow part

**Field 6**

Highest committed SDEP segment time stamp

**Field 7**

Sequential dependent High Water Mark

- Field 8**  
Highest committed SDEP segment
- Field 9**  
Logical begin time stamp

The following describes the contents of each of the nine fields in the table.

**Length (LL) (not shown in table)**

After a successful POS call, IMS places the length of the data area for this call in this 2-byte field.

**(Field 1)**

**Area name**

This 8-byte field contains the ddname from the AREA statement.

**Position**

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. These two fields uniquely identify a sequential dependent segment during the life of an area.

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 2)**

**Sequential dependent next to allocate CI**

This is the default if no keyword is specified as input in position one of the I/O area. The data returned is the 8-byte cycle count and RBA (CC+RBA) acquired from the global DMACNXTS field. This represents the next to be pre-allocated CI as a CI boundary.

**(Field 3)**

**Local sequential dependent next segment**

The data returned is the 8-byte CC+RBA of a segment boundary where the next SDEP to be inserted will be placed. This data is specific to only the IMS that executes the POS call. Its scope is for local IMS use only.

**(Field 4)**

**Unused CIs in sequential dependent part**

This 4-byte field contains the number of unused control intervals in the sequential dependent part.

**(Field 5)**

**Unused CIs in independent overflow part**

This 4-byte field contains the number of unused control intervals in the independent overflow part.

**(Field 6)**

**Highest committed SDEP segment time stamp**

The data returned is the 8-byte time stamp of the highest committed SDEP segment across partners, or for a local IMS, the time stamp of the pre-allocated SDEP dummy segment. If the area (either local or shared) has not been opened, or a /DBR was performed without any subsequent SDEP segment inserts, the current time is returned.

**(Field 7)**

**Sequential dependent High Water Mark**

This 8-byte field contains the cycle count plus RBA (CC+RBA) of the last pre-allocated CI which is the High Water Mark (HWM) CI.

**(Field 8)**

**Highest committed SDEP segment**

The data returned is the 8-byte cycle count plus RBA (CC+RBA) for the highest committed SDEP segment across partners, or for a local IMS, the CC+RBA of the highest committed SDEP segment. If the area (either local or shared) has not been opened, or a /DBR was performed without any subsequent SDEP segment inserts, the HWM CI is returned.

**(Field 9)**

**Logical begin time stamp**

This 8-byte field contains the logical begin time stamp from the DMACLBTS field.

*ssa*

Specifies the SSA that you want to use in this call. This parameter is an input parameter. The format of SSAs in POS calls is the same as the format of SSAs in DL/I calls. You can use only one SSA in this parameter. This parameter is optional for the POS call.

**Usage**

The POS call:

- Retrieves the location of a specific sequential dependent segment.
- Retrieves the location of last-inserted sequential dependent segment, its time stamp, and the IMS ID.
- Retrieves the time stamp of a sequential dependent segment or Logical Begin.
- Tells you 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.

If the area which the POS call specifies is unavailable, the I/O area is unchanged, and the status code FH is returned.

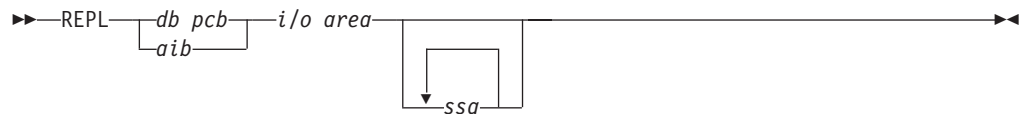
**Restrictions**

You can only use the POS call with a DEDB.

**REPL Call**

The Replace (REPL) call is used to change the values of one or more fields in a segment.

**Format**



	Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
For Full-Function:	REPL	X	X		X	
For DEDB:	REPL	X	X			
For MSDB:	REPL	X				

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the area in your program that communicates with IMS. This parameter is an input parameter. When you want to replace an existing segment in the database with a new segment, you first issue a Get Hold call to place the new segment in the I/O area. You can modify the data in the I/O area, and then issue the REPL call to replace the segment in the database.

### *ssa*

Specifies the SSAs, if any, to be used in the call. This parameter is an input parameter. The SSAs you supply in the call point to data areas in your program in which you have defined the SSAs for the call. You can use up to 15 SSAs in this parameter. This parameter is optional for the REPL call.

## Usage

A REPL call must be preceded by one of the three Get Hold calls. After you retrieve the segment, you modify it in the I/O area, and then issue a REPL call to replace it in the database. IMS replaces the segment in the database with the segment you modify in the I/O area. You cannot change the field lengths of the segments in the I/O area before you issue the REPL call.

For example, if you do not change one or more segments that are returned on a Get Hold call, or if you change the segment in the I/O area but do not want the change reflected in the database, you can inform IMS not to replace the segment. Specify an unqualified SSA with an N command code for that segment, which tells IMS not to replace the segment.

The N command enables you to tell IMS not to replace one or more of the multiple segments that were returned using the D command code. However, you can specify an N command code even if there were no D command codes on the preceding Get Hold call.

You should not include a qualified SSA on a REPL call. If you do, you receive an AJ status code.

For your program to successfully replace a segment, the segment must have been previously defined as replace-sensitive by PROCOPT=A or PROCOPT=R on the SENSEG statement in the PCB.

**Related Reading:** For more information on the PROCOPT option, see *IMS Version 9: Utilities Reference: Database and Transaction Manager*.

If your program attempts to do a path replace of a segment where it does not have replace sensitivity, and command code N is not specified, the data for the segment in the I/O area for the REPL call must be the same as the segment returned on the preceding Get Hold call. If the data changes in this situation, your program receives the status code, AM, and data does not change as a result of the REPL call.





---

## Chapter 5. Writing DL/I Calls for System Services

This chapter describes the calls you can use to obtain IMS DB system services for use in each type of application program, and the parameters for each call.

Each call description contains:

- A syntax diagram
- Definitions for parameters that are available to the call
- Details on how to use the call in your application program
- Restrictions on call usage, where applicable

Each parameter is described as an input parameter or output parameter. “Input” refers to input to IMS from the application program. “Output” refers to output from IMS to the application program.

Syntax diagrams for these calls begin with the *function* parameter. The call interface (xxxTDLI) and *parmcount* (if it is required) are not included in the syntax diagrams.

### **In this Chapter:**

- “APSB Call” on page 150
- “CHKP (Basic) Call” on page 150
- “CHKP (Symbolic) Call” on page 151
- “DPSB Call” on page 153
- “GMSG Call” on page 154
- “GSCD Call” on page 156
- “ICMD Call” on page 157
- “INIT Call” on page 159
- “INQY Call” on page 163
- “LOG Call” on page 169
- “PCB Call (CICS Online Programs Only)” on page 171
- “RCMD Call” on page 172
- “ROLB Call” on page 173
- “ROLL Call” on page 174
- “ROLS Call” on page 175
- “SETS/SETU Call” on page 176
- “SNAP Call” on page 177
- “STAT Call” on page 180
- “SYNC Call” on page 182
- “TERM Call (CICS Online Programs Only)” on page 183
- “XRST Call” on page 184

**Related Reading:** For specific information about coding your program in assembler language, C language, COBOL, Pascal, and PL/I, see Chapter 3, “Defining Application Program Elements,” on page 77 for the complete structure. For information on calls that apply to TM, see *IMS Version 9: Application Programming: Transaction Manager*. Calls within the section are in alphabetic order. For information on DL/I calls used for transaction management and EXEC DLI

commands used in CICS, see *IMS Version 9: Application Programming: Transaction Manager* and *IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS*.

---

## APSB Call

The Allocate PSB (APSB) calls are used to allocate a PSB for an ODBA application.

### Format

▶▶ APSB *aib* ◀◀

Call Name	DB/DC	IMS DB	DCCTL	DB Batch	TM Batch
APSB	X	X			

### Parameters

*aib*

Specifies the application interface block (AIB) that is used for the call. This parameter is an input and output parameter.

The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PSB name.

#### **AIBRSNM2**

This is the 4-character ID of ODBA startup table representing the target IMS of the APSB.

### Usage

The ODBA application must load or be link edited with the ODBA application interface AERTDLI.

The APSB call must be issued prior to any DLI calls.

The APSB call uses the AIB to allocate a PSB for ODBA application programs.

RRS/MVS must be active at the time of the APSB call. If RRS/MVS is not active, the APSB call will fail and the application will receive:

```
AIBRETRN = X'00000108'
AIBREASN = X'00000548'
```

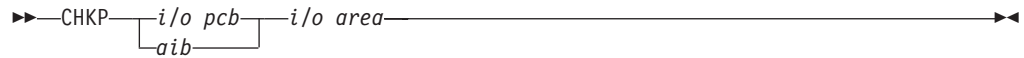
---

## CHKP (Basic) Call

A basic Checkpoint (CHKP) call is used for recovery purposes.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
CHKP	X	X	X	X	X

## Parameters

### *i/o pcb*

Specifies the I/O PCB for the call. A basic CHKP call must refer to the I/O PCB. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies your program's I/O area that contains the 8-byte checkpoint ID. This parameter is an input parameter. If the program is an MPP or a message-driven BMP, the CHKP call implicitly returns the next input message to this I/O area. Therefore, the area must be large enough to hold the longest returned message.

## Usage

Basic CHKP commits the changes your program has made to the database and establishes places in your program from which you can restart your program, if it terminates abnormally.

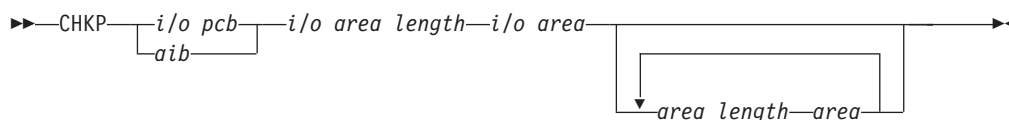
---

## CHKP (Symbolic) Call

A symbolic Checkpoint (CHKP) call is used for recovery purposes. If you use the symbolic Checkpoint call in your program, you also must use the XRST call.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
CHKP	X	X	X	X	X

## Parameters

### *i/o pcb*

Specifies the I/O PCB for the call. This parameter is an input and output parameter. A symbolic CHKP call must refer to the I/O PCB.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area length*

This parameter is no longer used by IMS. For compatibility reasons, this parameter must be included in the call, and it must contain a valid address. You can get a valid address by specifying the name of any area in your program.

### *i/o area*

Specifies the I/O area in your program that contains the 8-byte ID for this checkpoint. This parameter is an input parameter. If the program is a message-driven BMP, the CHKP call implicitly returns the next input message into this I/O area. Therefore, the area must be large enough to hold the longest returned message.

### *area length*

Specifies a 4-byte field in your program that contains the length (in binary) of the area to checkpoint. This parameter is an input parameter. You can specify up to seven area lengths. For each area length, you must also specify the *area* parameter. All seven *area* parameters (and corresponding length parameters) are optional. When you restart the program, IMS restores only the areas you specified in the CHKP call.

### *area*

Specifies the area in your program that you want IMS to checkpoint. This parameter is an input parameter. You can specify up to seven areas. Each area specified must be preceded by an *area length* parameter.

## Usage

The symbolic CHKP call commits the changes your program has made to the database and establishes places in your program from which you can restart your program, if it terminates abnormally. In addition, the CHKP call:

- Works with the Extended Restart (XRST) call to restart your program if it terminates abnormally
- Enables you to save as many as seven data areas in your program, which are restored when your program is restarted

An XRST call is required before a CHKP call to indicate to IMS that symbolic check points are being taken. The XRST call must specify a checkpoint ID of blanks. For more information, see “XRST Call” on page 184.

## Restrictions

The Symbolic CHKP call is allowed only from batch and BMP applications.

---

## DPSB Call

The DPSB call is used to deallocate IMS DB resources.

## Format

►►—DPSB—*aib*—◄◄

Call Name	DB/DC	IMS DB	DCCTL	DB Batch	TM Batch
DPSB	X	X			

## Parameters

*aib*

Specifies the application interface block (AIB) that is used for the call. This parameter is an input and output parameter.

The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PSB name.

**AIBSFUNC**

Subfunction code. This field must contain one of the 8-byte subfunction codes as follows:

bbbbbbbb (Null)  
PREPbbbb

## Usage

The DPSB call is used by an application running in a z/OS application region to deallocate a PSB. If the PREP subfunction is not used, the application must

activate sync-point processing prior to issuing the DPSB. Use the RRS/MVS SRRRCMIT/ATRCMIT calls to activate the sync-point process. Refer to *MVS Programming: Resource Recovery* for more information on these calls.

If the DPSB is issued before changes are committed, and, or locks released, the application will receive:

```
AIBRETRN = X'00000104'
AIBREASN = X'00000490'
```

The thread will not be terminated. The application should issue a SRRRCMIT or SRRBACK call, and retry the DPSB.

The PREP sub-function allows the application to issue the DPSB prior to activating the sync-point process. The sync-point activation can occur at a later time, but still must be issued.

## GMSG Call

A Get Message (GMSG) call is used in an automated operator (AO) application program to retrieve a message from the AO exit routine DFSAOE00.

## Format

```
▶▶ GMSG aib i/o area ▶▶
```

## Parameters

### *aib*

Specifies the application interface block (AIB) to be used for this call. This parameter is an input and output parameter.

You must initialize the following fields in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the length of the AIB the application actually obtained.

#### **AIBSFUNC**

Subfunction code. This field must contain one of the following 8-byte subfunction codes:

##### **8-blanks (null)**

When coded with an AOI token in the AIBRSNM1 field, indicates IMS is to return when no AOI message is available for the application program.

##### **WAITAOI**

When coded with an AOI token in the AIBRSNM1 field, WAITAOI indicates IMS is to wait for an AOI message when none is currently available for the application program. This subfunction value is invalid if an AOI token is not coded in AIBRSNM1. In this case, error return and reason codes are returned in the AIB.

The value WAITAOI must be left justified and padded on the right with a blank character.

**AIBRSNM1**

Resource name. This field must contain the AOI token or blanks. The AOI token identifies the message the AO application is to retrieve. The token is supplied for the first segment of a message. If the message is a multisegment message, set this field to blanks to retrieve the second through the last segment. AIBRSNM1 is an 8-byte alphanumeric left-justified field that is padded on the right with blanks.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list. This field is not changed by IMS.

**AIBOAUSE**

Length of the data returned in the I/O area. This parameter is an output parameter.

When partial data is returned because the I/O area is not large enough, AIBOAUSE contains the length required to receive all of the data, and AIBOALEN contains the actual length of the data.

*i/o area*

Specifies the I/O area to use for this call. This parameter is an output parameter. The I/O area should be large enough to hold the largest segment that is passed from IMS to the AO application program. If the I/O area is not large enough to contain all the data, IMS returns partial data.

## Usage

GMSG is used in an AO application program to retrieve a message associated with an AOI token. The AO application program must pass an 8-byte AOI token to IMS in order to retrieve the first segment of the message. IMS uses the AOI token to associate messages from AO exit routine DFSAOE00 with the GMSG call from an AO application program. IMS returns to the application program only those messages associated with the AOI token. By using different AOI tokens, DFSAOE00 can direct messages to different AO application programs. Note that your installation defines the AOI token.

**Related Reading:** For more information on the AOI exits, see *IMS Version 9: Customization Guide*.

To retrieve the second through the last segments of a multisegment message, issue GMSG calls with no token specified (set the token to blanks). If you want to retrieve all segments of a message, you must issue GMSG calls until all segments are retrieved. IMS discards all nonretrieved segments of a multisegment message when a new GMSG call that specifies an AOI token is issued.

Your AO application program can specify a wait on the GMSG call. If no messages are currently available for the associated AOI token, your AO application program waits until a message is available. The decision to wait is specified by the AO application program, unlike a WFI transaction where the wait is specified in the transaction definition. The wait is done on a call basis; that is, within a single application program some GMSG calls can specify waits, while others do not. Table 24 shows, by IMS environment, the types of AO application programs that can issue GMSG. GMSG is also supported from a CPI-C driven program.

Table 24. GMSG Support by Application Region Type

Application Region Type	IMS Environment		
	DBCTL	DB/DC	DCCTL
DRA thread	Yes	Yes	N/A
BMP (nonmessage-driven)	Yes	Yes	Yes
BMP (message-driven)	N/A	Yes	Yes
MPP	N/A	Yes	Yes
IFP	N/A	Yes	Yes

## Restrictions

A CPI-C driven program must issue an allocate PSB (APSB) call before issuing GMSG.

## GSCD Call

This section contains product-sensitive programming interface information.

A Get System Contents Directory (GSCD) call retrieves the address of the IMS system contents directory for batch programs.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
GSCD				X	X

## Parameters

### *db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

### *i/o pcb*

Specifies the I/O PCB for the call. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.



**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb (if the I/O PCB is used), or the name of a DB PCB (if a DB PCB is used).

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the I/O area, which must be 8 bytes long. IMS places the address of the system contents directory (SCD) in the first 4 bytes and the address of the program specification table (PST) in the second 4 bytes. This parameter is an output parameter.

**Usage**

IMS does not return a status code to a program after it issues a successful GSCD call. The status code from the previous call that used the same PCB remains unchanged in the PCB. For more information on GSCD, see *IMS Version 9: Application Programming: Design Guide*.

**Restriction**

The GSCD call can be issued only from batch application programs.

**ICMD Call**

An Issue Command (ICMD) call enables an automated operator (AO) application program to issue an IMS command and retrieve the first command response segment.

**Format**

▶▶—ICMD—*aib*—*i/o area*—————▶▶

**Parameters***aib*

Specifies the application interface block (AIB) for this call. This parameter is an input and output parameter.

The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list. This field is not changed by IMS.

**AIBOAUSE**

Length of data returned in the I/O area. This parameter is an output parameter.

Your program must check this field to determine whether the ICMD call returned data to the I/O area. When the only response to the command is a DFS058 message indicating that the command is either in progress or complete, the response is not returned.

When partial data is returned because the I/O area is not large enough, AIBOAUSE contains the length required to receive all of the data, and AIBOALEN contains the actual length of the data.

#### *i/o area*

Specifies the I/O area to use for this call. This parameter is an input and output parameter. The I/O area should be large enough to hold the largest command that is passed from the AO application program to IMS, or the largest command response segment that is passed from IMS to the AO application program. If the I/O area is not large enough to contain all the data, IMS returns partial data.

## Usage

ICMD enables an AO application to issue an IMS command and retrieve the first command response segment.

When using ICMD, put the IMS command that is to be issued in your application program's I/O area. After IMS has processed the command, it returns the first segment of the response message to your AO application program's I/O area. To retrieve subsequent segments (one segment at a time) use the RCMD call.

Some IMS commands that complete successfully result in a DFS058 message indicating that the command is complete. Some IMS commands that are processed asynchronously result in a DFS058 message indicating that the command is in progress. For a command entered on an ICMD call, neither DFS058 message is returned to the AO application program. In this case, the AIBOAUSE field is set to 0 to indicate that no segment was returned. So, your AO application program must check the AIBOAUSE field along with the return and reason codes to determine if a response was returned.

**Related Reading:** For more information on the AOI exits, see *IMS Version 9: Customization Guide*.

Table 25 shows, by IMS environment, the types of AO application programs that can issue ICMD. ICMD is also supported from a CPI-C driven program.

Table 25. ICMD Support by Application Region Type

Application Region Type	IMS Environment		
	DBCTL	DB/DC	DCCTL
DRA thread	Yes	Yes	N/A
BMP (nonmessage-driven)	Yes	Yes	Yes
BMP (message-driven)	N/A	Yes	Yes
MPP	N/A	Yes	Yes
IFP	N/A	Yes	Yes

See *IMS Version 9: Command Reference* for a list of commands that can be issued using the ICMD call.

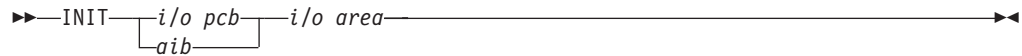
## Restrictions

Before issuing ICMD, a CPI-C driven program must issue an allocate PSB (APSB) call.

## INIT Call

The Initialize (INIT) call allows an application to receive status codes regarding deadlock occurrences and data availability (by checking each DB PCB).

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
INIT	X	X	X	X	X

## Parameters

### *i/o pcb*

Specifies the I/O PCB for the call. INIT must refer to the I/O PCB. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the I/O area in your program that contains the character string or strings indicating which INIT functions are requested. This parameter is an input parameter. INIT function character strings include DB QUERY, STATUS GROUPA, and STATUS GROUPB.

## Usage

You can use the call in any application program, including IMS batch in a sharing environment.

Specify the function in your application program with a character string in the I/O area.

**Example:** Use the format LLZZCharacter-String, where LL is the length of the character string including the length of the LLZZ portion; ZZ must be binary 0. For PL/I, you must define the LL field as a fullword; the value is the length of the character string including the length of the LLZZ portion, minus 2. If the I/O area is invalid, an AJ status code is returned. Table 28 on page 161 and Table 29 on page 161 contain sample I/O areas for INIT when it is used with assembler language, COBOL, C language, Pascal, and PL/I.

### Determining Database Availability: INIT DBQUERY

When the INIT call is issued with the DBQUERY character string in the I/O area, the application program can obtain information regarding the availability of data for each PCB. Table 26 contains a sample I/O area for the INIT call with DBQUERY for assembler language, COBOL, C language, and Pascal.

Table 26. INIT DBQUERY: Examples for ASMTDLI, CBLTDLI, CTDLI, and PASTDLI

L	L	Z	Z	Character String
00	0B	00	00	DBQUERY

**Note:** The LL value of X'0B' is a hexadecimal representation of decimal 11. ZZ fields are binary.

Table 27 contains a sample I/O area for the INIT call with DBQUERY for PL/I.

Table 27. INIT DBQUERY: I/O Area Example for PLITDLI

L	L	L	L	Z	Z	Character String
00	00	00	0B	00	00	DBQUERY

**Note:** The LL value of X'0B' is a hexadecimal representation of decimal 11. ZZ fields are binary.

**LL or LLLL** A 2-byte field that contains the length of the character string, plus two bytes for LL. For the PLITDLI interface, use the 4-byte field LLLL. When you use the AIB interface (AIBTDLI), PL/I programs require only a 2-byte field.

**ZZ** A 2-byte field of binary zeros.

One of the following status codes is returned for each database PCB:

**NA** At least one of the databases that can be accessed using this PCB is not available. A call made using this PCB probably results in a BA or BB status code if the INIT STATUS GROUPA call has been issued, or in a DFS3303I message and 3303 pseudoabend if it has not. An exception is when the database is not available because dynamic allocation failed. In this case, a call results in an AI (unable to open) status code.

In a DCCTL environment, the status code is always NA.

**NU** At least one of the databases that can be updated using this PCB is unavailable for update. An ISRT, DLET, or REPL call using this PCB might result in a BA status code if the INIT STATUS GROUPA call has been issued, or in a DFS3303I message and 3303 pseudoabend if it has not. The database that caused the NU status code might be required only for delete processing. In that case, DLET calls fail, but ISRT and REPL calls succeed.

**bb** The data that can be accessed with this PCB can be used for all functions that the PCB allows. DEDBs and MSDBs always have the bb status code.

In addition to data availability status, the name of the database organization of the root segment is returned in the segment name field of the PCB. The segment name field contains one of the following database organizations: DEDB, MSDB, GSAM, HDAM, PHDAM, HIDAM, PHIDAM, HISAM, HSAM, INDEX, SHSAM, or SHISAM.

For a DCCTL environment, the database organization is UNKNOWN.

**Important:** If you are working with a High Availability Large Database (HALDB), you need to be aware that the feedback on data availability at PSB schedule time only shows the availability of the HALDB master, not of the HALDB partitions. However, the error settings for data unavailability of a HALDB partition are the same as those of a non-HALDB database, namely status code 'BA' or pseudo abend U3303.

**Related Reading:** For more information on HALDB, see “High Availability Large Databases” on page 18.

**Automatic INIT DBQUERY**

When the program is initially scheduled, the status code in the database PCBs is initialized as if the INIT DBQUERY call were issued. The application program can therefore determine database availability without issuing the INIT call.

For a DCCTL environment, the status code is NA.

**Performance Considerations for the INIT Call (IMS Online Only)**

For performance reasons, the INIT call should not be issued before the first GU call to the I/O PCB. If the INIT call is issued first, the GU call is not processed as efficiently.

**Enabling Data Availability Status Codes: INIT STATUS GROUPA**

Table 28 contains a sample I/O area for the INIT call for assembler language, COBOL, C language, and Pascal.

*Table 28. INIT I/O Area Examples for ASMTDLI, CBLTDLI, CTDLI, and PASTDLI*

L	L	Z	Z	Character String
00	11	00	00	STATUS GROUPA

**Note:** The LL value of X'11' is a hexadecimal representation of decimal 17. ZZ fields are binary.

Table 29 contains a sample I/O area for the INIT call for PL/I.

*Table 29. INIT I/O Area Examples for PLITDLI*

L	L	L	L	Z	Z	Character String
00	00	00	11	00	00	STATUS GROUPA

**Note:** The LL value of X'11' is a hexadecimal representation of decimal 17. ZZ fields are binary.

**LL or LLLL** LL is a halfword-length field. For non-PLITDLI calls, LLLL is a fullword-length field for PLITDLI.

**ZZ** A 2-byte field of binary zeros.

The value for LLZZ data or LLLLZZ data is always 4 bytes (for LLZZ or LLLLZZ), plus data length.

**Recommendation:** You should be familiar with data availability.

**Related Reading:** For more information about data availability, see *IMS Version 9: Application Programming: Design Guide*.

When the INIT call is issued with the character string STATUS GROUPA in the I/O area, the application program informs IMS that it is prepared to accept status codes regarding data unavailability. IMS then returns a status code rather than a resultant pseudoabend if a subsequent call requires access to unavailable data. The status codes that are returned when IMS encounters unavailable data are BA and BB. Status codes BA and BB both indicate that the call could not be completed because it required access to data that was not available. DEDBs can receive the BA or BB status code.

In response to status code BA, the system backs out only the updates that were done for the current call before it encountered the unavailable data. If changes have been made by a previous call, the application must decide to commit or not commit to these changes. The state of the database is left as it was before the failing call was issued. If the call was a REPL or DLET call, the PCB position is unchanged. If the call is a Get type or ISRT call, the PCB position is unpredictable.

In response to status code BB, the system backs out all database updates that the program made since the last commit point and cancels all nonexpress messages that were sent since the last commit point. The PCB position for all PCBs is at the start of the database.

### Enabling Deadlock Occurrence Status Codes: INIT STATUS GROUPB

Table 30 contains a sample I/O area for the INIT call for assembler language, COBOL, C language, and Pascal.

Table 30. INIT I/O Area Examples for ASMTDLI, CBLTDLI, CTDLI, and PASTDLI

L	L	Z	Z	Character String
00	11	00	00	STATUS GROUPB

**Note:** The LL value of X'11' is a hexadecimal representation of decimal 17. ZZ fields are binary.

Table 31 contains a sample I/O area for the INIT call for PL/I.

Table 31. INIT I/O Area Examples for PLITDLI

L	L	L	L	Z	Z	Character String
00	00	00	11	00	00	STATUS GROUPB

**Note:** The LL value of X'11' is a hexadecimal representation of decimal 17. ZZ fields are binary.

**LL or LLLL** LL is a halfword-length field. For non-PLITDLI calls, LLLL is a fullword-length field for PLITDLI.

**ZZ** A 2-byte field of binary zeros.

The value for LLZZ data or LLLLZZ data is always four bytes (for LLZZ or LLLLZZ), plus data length.

When the INIT call is issued with the character string STATUS GROUPB in the I/O area, the application program informs IMS that it is prepared to accept status codes regarding data unavailability and deadlock occurrences. The status codes for data unavailability are BA and BB, as described under “Enabling Data Availability Status Codes: INIT STATUS GROUPA” on page 161.

When a deadlock occurs in batch and the INITSTATUS GROUPB call has been issued, the following occurs:

- If no changes were made to the database, the BC status code is returned.
- If updates were made to the database, and if a datalog exists and BKO=YES is specified, the BC status code is returned.
- If changes were made to the database, and a disklog does not exist or BKO=YES is not specified, a 777 pseudoabend occurs.

When the application program encounters a deadlock occurrence, IMS:

- Backs out all database resources (with the exception of GSAM and DB2) to the last commit point. Although GSAM PCBs can be defined for pure batch or BMP environments, GSAM changes are not backed out. Database resources are backed out for DB2 only when IMS is the sync-point coordinator.

When you use INIT STATUS GROUPB in a pure batch environment, you must specify the DISKLOG and BACKOUT options.

- Backs out all output messages to the last commit point.
- Requeues all input messages as follows:

<b>Environment</b>	<b>Action</b>
<b>MPP and BMP</b>	All input messages are returned to the message queue. The application program no longer controls its input messages.
<b>IFP</b>	All input messages are returned to IMS Fast Path (IFP) balancing group queues (BALGRP), making them available to any other IFP region on the BALGRP. The IFP that is involved in the deadlock receives the next transaction or message that is available on the BALGRP.
<b>DBCTL</b>	Action is limited to resources that are managed by DBCTL, for example, database updates.

- Returns a BC status code to the program in the database PCB.

## Restrictions

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

You should be familiar with deadlock occurrences as described in *IMS Version 9: Administration Guide: System*.

---

## INQY Call

The Inquiry (INQY) call is used to request information regarding execution environment, destination type and status, and session status. INQY is valid only when using the AIB interface.

## Format

►►—INQY—*aib*—*i/o area*—►►

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
INQY	X	X	X	X	X

## Parameters

### *aib*

Specifies the address of the application interface block (DFSAIB) for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIB**bb**.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBSFUNC**

Subfunction code. This field must contain one of the 8-byte subfunction codes as follows:

- DBQUERY**b**
- ENVIRON**b**
- FIND**bbbb**
- LERUNOPT
- PROGRAM**b**

Not supported with the ODBA interface.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of any named PCB in the PSB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list. This field is not changed by IMS.

### *i/o area*

Specifies the data output area to use with the call. This parameter is an output parameter. An I/O area is required for INQY subfunctions ENVIRON**b** and PROGRAM**b**. It is not required for subfunctions DBQUERY**b** and FIND**bbbb**.

## Usage

The INQY call operates in both batch and online IMS environments. IMS application programs can use the INQY call to request information regarding the output destination, the session status, the current execution environment, the availability of databases, and the PCB address, which is based on the PCB name. You must use the AIB when issuing an INQY call. Before you can issue an INQY call, initialize the fields of the AIB. For more information on initializing AIBs, see “The AIBTDLI Interface” on page 111.



When you use the INQY call, specify an 8-byte subfunction code, which is passed in the AIB. The INQY subfunction determines the information that the application program receives. For a summary of PCB type and I/O area use for each subfunction, see Table 33 on page 169.

The INQY call returns information to the caller's I/O area. The length of the data that is returned from the INQY call is passed back to the application program in the AIB field, AIBOAUSE.

You specify the size of the I/O area in the AIB field, AIBOALEN. The INQY call returns only as much data as the area can hold in one call. If the area is not large enough for all the information, an AG status code is returned, and partial data is returned in the I/O area. In this case, the AIB field AIBOALEN contains the actual length of the data returned to the I/O area, and the AIBOAUSE field contains the output area length that would be required to receive all the data.

### Querying Data Availability: INQY DBQUERY

When the INQY call is issued with the DBQUERY subfunction, the application program obtains information regarding the data for each PCB. The only valid PCB name that can be passed in AIBRSNM1 is IOPCBbbb. The INQY DBQUERY call is similar to the INITDBQUERY call. The INIT DBQUERY call does not return information in the I/O area, but like the INIT DBQUERY call, it updates status codes in the database PCBs.

In addition to the INIT DBQUERY status codes, the INQY DBQUERY call returns these status codes in the I/O PCB:

- bb** The call is successful and all databases are available.
- BJ** None of the databases in the PSB are available, or no PCBs exist in the PSB. All database PCBs (excluding GSAM) contain an NA status code as the result of processing the INQY DBQUERY call.
- BK** At least one of the databases in the PSB is not available or availability is limited. At least one database PCB contains an NA or NU status code as the result of processing the INQY DBQUERY call.

The INQY call returns the following status codes in each DB PCB:

- NA** At least one of the databases that can be accessed using this PCB is not available. A call that is made using this PCB probably results in a BA or BB status code if the INIT STATUS GROUPA call has been issued, or in a DFS3303I message and 3303 pseudoabend if the call has not been issued. An exception is when the database is not available because dynamic allocation failed. In this case, a call results in an AI (unable to open) status code.

In a DCCTL environment, the status code is always NA.

- NU** At least one of the databases that can be updated using this PCB is unavailable for update. An ISRT, DLET, or REPL call using this PCB might result in a BA status code if the INIT STATUS GROUPA call has been issued, or in a DFS3303I message and 3303 pseudoabend if it has not been issued. The database that caused the NU status code might be required only for delete processing. In that case, DLET calls fail, but ISRT and REPL calls succeed.
- bb** The data that can be accessed with this PCB can be used for all functions the PCB allows. DEDBs and MSDBs always have the bb status code.

### Querying the Environment: INQY ENVIRON

When the INQY call is issued with the ENVIRON subfunction, the application program obtains information regarding the current execution environment. The only valid PCB name that can be passed in AIBRSNM1 is IOPCBbbb. This includes the IMS identifier, release, region, and region type.

The INQY ENVIRON call returns character-string data. The output is left justified and padded with blanks on the right.

**Recommendation:** To receive the following data and to account for expansion, define the I/O area length to be larger than 152 bytes. If you define the I/O area length to be exactly 152 bytes and the I/O area is expanded in future releases, you will receive an AG status code.

100 bytes	INQY ENVIRON data
2 bytes	Length field for Recovery Token section (18 bytes)
16 bytes	Recovery Token
2 bytes	Length field for APARM section (maximum of 34 bytes)
32 bytes	APARM data
<hr/>	
152 bytes	Total I/O area length

Table 32 lists the output that is returned from the INQY ENVIRON call. Included with the information returned is the output's byte length, the actual value, and an explanation.

Table 32. INQY ENVIRON Data Output

Information Returned	Length in Bytes	Actual Value	Explanation
IMS Identifier	8		Provides the identifier from the execution parameters.
IMS Release Level	4		Provides the release level for IMS. For example, X'00000410'.
IMS Control Region Type	8	BATCH	Indicates that an IMS batch region is active.
		DB	Indicates that only the IMS Database Manager is active. (DBCTL system)
		TM	Indicates that only the IMS Transaction Manager is active. (DCCTL system)
		DB/DC	Indicates that both the IMS Database and Transaction managers are active. (DBDC system)
IMS Application Region Type	8	BATCH	Indicates that the IMS Batch region is active.
		BMP	Indicates that the Batch Message Processing region is active.
		DRA	Indicates that the Database Resource Adapter Thread region is active.
		IFP	Indicates that the IMS Fast Path region is active.
		MPP	Indicates that the Message Processing region is active.
Region Identifier	4		Provides the region identifier. For example, X'00000001'.
Application Program Name	8		Provides the name of the application program being run.
PSB Name (currently allocated)	8		Provides the name of the PSB currently allocated.
Transaction Name	8		Provides the name of the transaction.
		b	Indicates that no associated transaction exists.

Table 32. INQY ENVIRON Data Output (continued)

Information Returned	Length in Bytes	Actual Value	Explanation
User Identifier <sup>1</sup>	8		Provides the user ID.
		b	Indicates that the user ID is unavailable.
Group Name	8		Provides the group name.
		b	Indicates that the group name is unavailable.
Status Group Indicator	4	A	Indicates an INIT STATUS GROUPA call is issued.
		B	Indicates an INIT STATUS GROUPB call is issued.
		b	Indicates that a status group is not initialized.
Address of Recovery Token <sup>2</sup>	4		Provides the address of the LL field, followed by the recovery token.
Address of the Application Parameter String <sup>2</sup>	4		Provides the address of the LL field, followed by the application program parameter string.
		0	Indicates that the APARM= parameter is not coded in the execution parameters of the dependent region JCL.
Shared Queues Indicator	4		Indicates IMS is not using Shared Queues.
		SHRQ	Indicates IMS is using Shared Queues.
Userid of Address Space	8		Userid of dependent address space.
Userid Indicator	1		The Userid Indicator field has one of four possible values. This value indicates the contents of the userid field.
		U	Indicates the user's identification from the source terminal during sign-on.
		L	Indicates the LTERM name of the source terminal in sign-on is not active.
		P	Indicates the PSBNAME of the source BMP or transaction.
		O	Indicates some other name.
			3
RRS Indicator	3	b	Indicates IMS has not expressed interest in the UR with RRS. Therefore, the application should refrain from performing any work that causes RRS to become the syncpoint manager for the UR because IMS will not be involved in the commit scope. For example, the application should not issue any outbound protected conversations.
		RRS	Indicates IMS has expressed interest in the UR with RRS. Therefore, IMS will be involved in the commit scope if RRS is the syncpoint manager for the UR.

**Notes:**

- The user ID is derived from the PSTUSID field of the PST that represents the region making the INQY ENVIRON call. The PSTUSID field is one of the following:
  - For message-driven BMP regions that have not completed successful GU calls to the IMS message queue and for non-message-driven BMP regions, the PSTUSID field is derived from the name of the PSB that is currently scheduled into the BMP region.
  - For message-driven BMP regions that have completed a successful GU call and for any MPP region, the PSTUSID field is derived which is usually the input terminal's RACF ID. If the terminal has not signed on to RACF, the ID is the input terminal's LTERM.
- The pointer identifies a length field (LL) that contains the length of the recovery token or application program parameter string in binary, including the two bytes required for LL.

### Querying the PCB: INQY FIND

When the INQY call is issued with the FIND subfunction, the application program is returned with the PCB address of the requested PCB name. The only valid PCB names that can be passed in AIBRSNM1 are IOPCBbbb or the name of an alternate PCB or DB PCB, as defined in the PSB.

On a FIND subfunction, the requested PCB remains unmodified, and no information is returned in an I/O area.

The FIND subfunction is used to get a PCB address following an INQY DBQUERY call. This process allows the application program to analyze the PCB status code to determine if either an NA or NU status code is set in the PCB.

### Querying for LE Overrides: INQY LERUNOPT

When the LERUNOPT call is issued with the LERUNOPT subfunction, IMS determines if LE overrides are allowed based on the LEOPT system parameter. The LE override parameters are defined to IMS through the UPDATE LE command. IMS checks to see if there are any overrides applicable to the caller based on the specific combinations of transaction name, lterm name, userid, or program name in the caller's environment. IMS will return the address of the string to the caller if an override parameter is found. The LE overrides are used by the IMS supplied CEEBXITA exit, DFSBXITA, to allow dynamic overrides for LE runtime parameters.

#### Related Reading:

- For more information about the UPDATE LE command, see *IMS Version 9: Command Reference*.
- For more information about the IMS supplied CEEBXITA, DFSBXITA, see *IMS Version 9: Customization Guide*.

The call string must contain the function code and the AIB address. The I/O area is not a required parameter and will be ignored if specified. The only valid PCB name that can be passed in AIBRSNM1 is IOPCB. The AIBOALEN and AIBOAUUSE fields are not used.

The rules for matching an entry that results in it being returned on a DL/I INQY LERUNOPT call are:

- An MPP or JMP region uses transaction name, lterm, userid, and program to match with each entry.
- An IFB, JBP, or non-message driven BMP uses program name to match with each entry. If an entry has a defined filter for transaction name, lterm, or userid, it does not match. Message driven BMPs also use transaction name.
- The entries are scanned to find the entry with the most filter matches. The first entry in the list with the most exact filter matches is selected. The scan stops with an entry found with all of the filters matching the entry.

**Note:** Searching table entries may cause user confusion because of the way entries are built and searched. For example, assume there are two entries in the table that match on the filters specified on the DL/I INQY call. The first transaction matches on transaction name and lterm name. The second entry matches on transaction name and program name. IMS chooses the first entry because it was the first entry encountered with highest number of filter matches. If the second entry is now updated with a longer parameter string, which causes a new entry to be built, it will be added to the head of the queue. The next search would result in the entry

with transaction name and program name being selected. This could result in a set of runtime options being selected that were not expected by the user.

**Querying the Program Name: INQY PROGRAM**

When you issue the INQY call with the PROGRAM subfunction, the application program name is returned in the first 8 bytes of the I/O area. The only valid PCB name that can be passed in AIBRSNM1 is IOPCBbbb.

**INQY Return Codes and Reason Codes**

When you issue the INQY call, return and reason codes are returned to the AIB. Status codes can be returned to the PCB. If return and reason codes other than those that apply to INQY are returned, your application should examine the PCB to see what status codes are found.

**Map of INQY Subfunction to PCB Type**

*Table 33. Subfunction, PCB, and I/O Area Combinations for the INQY Call*

Subfunction	I/O PCB	Alternate PCB	DB PCB	I/O Area Required
FIND	OK	OK	OK	NO
ENVIRON	OK	NO	NO	YES
DBQUERY	OK	NO	NO	NO
LERUNOPT	OK	NO	NO	NO
PROGRAM	OK	NO	NO	YES

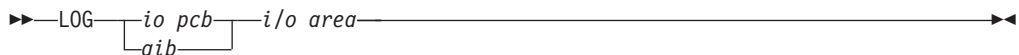
**Restrictions**

The INQY call is valid only when using the AIB. An INQY call issued through the PCB interface is rejected with an AD status code.

**LOG Call**

The Log (LOG) call is used to send and write information to the IMS system log.

**Format**



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
LOG	X	X	X	X	X

**Parameters**

*io pcb*

Specifies the I/O PCB for the call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the area in your program that contains the record that you want to write to the system log. This is an input parameter. This record must follow the format shown in Table 34 and Table 35. The format of this record is described in more detail following the tables.

Table 34. Log Record Formats for COBOL, C, Assembler, Pascal, and PL/I Programs for the AIBTDLI, ASMTDLI, CBLTDLI, CEETDLI, CTDLI, and PASTDLI Interfaces

LL	ZZ	C	Text
2	2	1	Variable

Table 35. Log Record Formats for COBOL, C, Assembler, Pascal, and PL/I Programs for the PLITDLI Interface

LLLL	ZZ	C	Text
4	2	1	Variable

The fields must be:

- LL or LLLL** Specifies a 2-byte field (or, for PL/I, a 4-byte-long field) to contain the length of the record. The length of the record is equal to LL + ZZ + C + text of the record. When you calculate the length of the log record, you must account for all fields. The total length you specify includes:
- 2 bytes for LL or LLLL. (For PL/I, include the length as 2, even though LLLL is a 4-byte field.)
  - 2 bytes for the ZZ field.
  - 1 byte for the C field.
  - *n* bytes for the length of the record itself.

If you are using the PLITDLI interface, your program must define the length field as a binary fullword.

**ZZ**

Specifies a 2-byte field of binary zeros.

**C**

Specifies a 1-byte field containing a log code, which must be equal to or greater than X'A0'.

**Text**

Specifies any data to be logged.

## Usage

An application program can write a record to the system log by issuing the LOG call. When you issue the LOG call, specify the I/O area that contains the record you want

written to the system log. You can write any information to the log, and you can use different log codes to distinguish between different types of information.

You can issue the LOG call:

- In a batch program, and the record is written to the IMS log
- In an online program in the DBCTL environment, and the record is written to the DBCTL log
- In the IMS DB/DC environment, and the record is written to the IMS log

## Restrictions

The length of the I/O area (including all fields) cannot be larger than the logical record length (LRECL) for the system log data set, minus four bytes, or the I/O area specified in the IOASIZE keyword of the PSBGEN statement of the PSB.

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

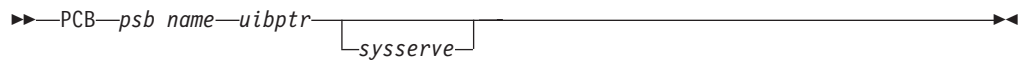
---

## PCB Call (CICS Online Programs Only)

The PCB call is used to schedule a PSB call.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
PCB	X	X			

## Parameters

The AIB is not valid for PCB calls.

### *psb name*

Specifies the PSB. An asterisk can be used for the parameter to indicate the default. This parameter is an input parameter.

### *uibptr*

Specifies a pointer, which is set to the address of the UIB after the call. This parameter is an output parameter.

### *sysserve*

Specifies an optional 8-byte field that contains either IOPCB or NOIOPCB. This parameter is an input parameter.

## Usage

Before a CICS online program can issue any DL/I calls, it must indicate to DL/I its intent to use a particular PSB. A PCB call accomplishes this and also obtains the address of the PCB list in the PSB. When you issue a PCB call, specify the following:

- The call function: PCBb

- The PSB you want to use, or an asterisk to indicate that you want to use the default name. The default PSB name is not necessarily the name of the program issuing the PCB call, because that program could have been called by another program.
- A pointer, which is set to the address of the UIB after the call.  
For more information on defining and establishing addressability to the UIB, see “Specifying the UIB (CICS Online Programs Only)” on page 102.
- The system service call parameter that names an optional 8-byte field that contains either IOPCB or NOIOPCB.

## Restrictions

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

---

## RCMD Call

A Retrieve Command (RCMD) call enables an automated operator (AO) application program retrieve the second and subsequent command response segments after an ICMD call.

## Format

►►—RCMD—*aib*—*i/o area*—◄◄

## Parameters

### *aib*

Specifies the application interface block (AIB) used for this call. This parameter is an input and output parameter.

The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list. This field is not changed by IMS.

#### **AIBOAUSE**

Length of data returned in the I/O area. This parameter is an output parameter.

When partial data is returned because the I/O area is not large enough, AIBOAUSE contains the length required to receive all of the data, and AIBOALEN contains the actual length of the data.

### *i/o area*

Specifies the I/O area to use for this call. This parameter is an output parameter. The I/O area should be large enough to hold the largest command response segment that is passed from IMS to the AO application program. If the I/O area is not large enough for all of the information, partial data is returned in the I/O area.



## Usage

RCMD lets an AO application program retrieve the second and subsequent command response segments resulting from an ICMD call.

**Related Reading** For more information on the AOI exits, see *IMS Version 9: Customization Guide*.

Table 36 shows, by IMS environment, the types of AO application programs that can issue RCMD. RCMD is also supported from a CPI-C driven program.

Table 36. RCMD Support by Application Region Type

Application Region Type	IMS Environment		
	DBCTL	DB/DC	DCCTL
DRA thread	Yes	Yes	N/A
BMP (nonmessage-driven)	Yes	Yes	Yes
BMP (message-driven)	N/A	Yes	Yes
MPP	N/A	Yes	Yes
IFP	N/A	Yes	Yes

RCMD retrieves only one response segment at a time. If you need additional response segments, you must issue RCMD one time for each response segment that is issued by IMS.

## Restrictions

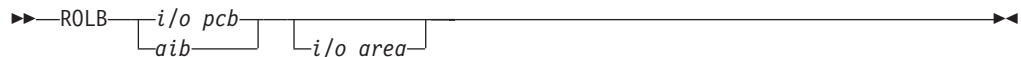
An ICMD call must be issued before an RCMD call.

## ROLB Call

The Roll Back (ROLB) call is used to dynamically back out database changes and return control to your program. For more information on the ROLB call, see “Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)” on page 250.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
ROLB	X	X	X	X	X

## Parameters

### *i/o pcb*

Specifies the I/O PCB for the call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the area in your program where IMS returns the first message segment. This parameter is an output parameter.

## Restrictions

The AIB must specify the I/O PCB for this call.

---

## ROLL Call

The Roll (ROLL) call is used to abnormally terminate your program and to dynamically back out database changes. For more information on the ROLL call, see "Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)" on page 250.

The ODBA interface does not support this call.

## Format

►►—ROLL—◄◄

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
ROLL	X	X	X	X	X

## Parameters

The only parameter required for the ROLL call is the call function.

## Usage

When you issue a ROLL call, IMS terminates the application program with a U0778abend.

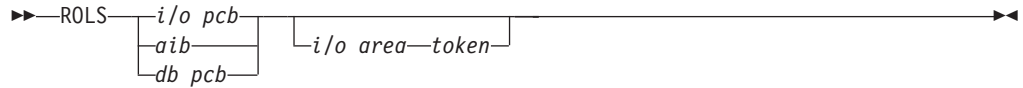
## Restriction

Unlike the ROLB call, the ROLL call does not return control to the program.

## ROLS Call

The Roll Back to SETS (ROLS) call is used to back out to a processing point set by a prior SETS or SETU call. For more information on the ROLS call, see “Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)” on page 250.

### Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
ROLS	X	X	X	X	X

### Parameters

*db pcb*

Specifies the DB PCB for the call. This parameter is an input and output parameter.

*i/o pcb*

Specifies the I/O PCB for the call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb, or the name of a DB PCB.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the I/O area has the same format as the I/O area supplied on the SETS call. This parameter is an output parameter.

*token*

Specifies the area in your program that contains a 4-byte identifier. This parameter is an input parameter.

### Usage

When you use the Roll Back to SETS (ROLS) call to back out to a processing point set by a prior SETS or SETU, the ROLS enables you to continue processing or to back out to the prior commit point and place the input message on the suspend queue for later processing.

Issuing a ROLS call for a DB PCB can result in the user abend code 3303.

## Restrictions

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

The ROLS call is not valid when the PSB contains a DEDB or MSDB PCB, or when the call is made to a DB2 database.

---

## SETS/SETU Call

The Set a Backout Point (SETS) call is used to set an intermediate backout point or to cancel all existing backout points. The SET Unconditional (SETU) call operates like the SETS call, except that the SETU call is accepted even if unsupported PCBs exist or an external subsystem is used. For more information on the SETS and SETU calls, see "Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)" on page 250.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
SETS/SETU	X	X	X	X	X

## Parameters

### *i/o pcb*

Specifies the I/O PCB for the call. SETS and SETU must refer to the I/O PCB. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies the area in your program that contains the data to be returned on the corresponding ROLS call. This parameter is an input parameter.

*token*

Specifies the area in your program that contains a 4-byte identifier. This parameter is an input parameter.

## Usage

The SETS and SETU format and parameters are the same, except for the call functions, SETS and SETU.

The SETS and SETU calls provide the backout points that IMS uses in the ROLS call. The ROLS call operates with the SETS and SETU call backout points.

The meaning of the SC status code for SETS and SETU is as follows:

**SETS** The SETS call is rejected. The SC status code in the I/O PCB indicates that either the PSB contains unsupported options or the application program made calls to an external subsystem.

**SETU** The SETU call is not rejected. The SC status code indicates either that unsupported PCBs exist in the PSB or the application program made calls to an external subsystem.

## Restrictions

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

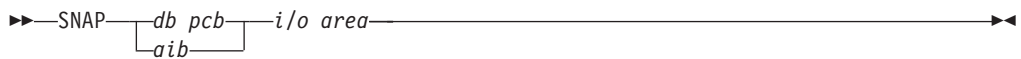
The SETS call is not valid when the PSB contains a DEDB or MSDB PCB, or when the call is made to a DB2 database. The SETU call is valid, but not functional, if unsupported PCBs exist in the PSB or if the program uses an external subsystem.

## SNAP Call

This section contains product-sensitive programming interface information.

The SNAP call is used to collect diagnostic information.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
SNAP	X	X		X	

## Parameters

*db pcb*

Specifies the address that refers to a full-function PCB that is defined in a calling program PSB. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a full-function DB PCB.

**AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

*i/o area*

Specifies the area in your program that contains SNAP operation parameters. This parameter is an input parameter. Figure 33 shows the SNAP operation parameters you specify, including:

- Length for bytes 1 through 2
- Destination for bytes 3 through 10
- Identification for bytes 11 through 18
- SNAP options for bytes 19 through 22

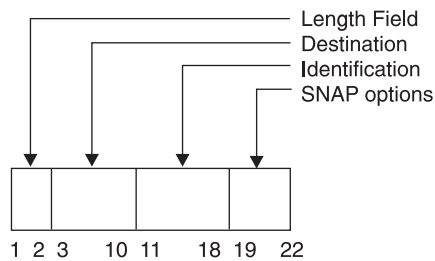


Figure 33. I/O Area for SNAP Operation Parameters

Table 37 explains the values that you can specify.

Table 37. SNAP Operation Parameters

Byte	Value	Meaning															
1-2	xx	<p>This 2-byte binary field specifies the length of the SNAP operation parameters. The length must include this 2-byte length field.</p> <p>When you do not specify operation parameters, IMS uses default values. The following chart lists the lengths that result from your parameter specifications.</p> <table border="1"> <thead> <tr> <th>If you supply values for:</th> <th>And IMS supplies default values for:</th> <th>Then the length (in hexadecimal) is:</th> </tr> </thead> <tbody> <tr> <td>Destination, Identification, SNAP options</td> <td></td> <td>16</td> </tr> <tr> <td>Destination, Identification</td> <td>SNAP options</td> <td>12</td> </tr> <tr> <td>Destination</td> <td>Identification, SNAP options</td> <td>10</td> </tr> <tr> <td></td> <td>Destination, Identification, SNAP options</td> <td>2</td> </tr> </tbody> </table> <p>If you specify another length, IMS uses default values for the destination, identification, and SNAP operation parameters.</p>	If you supply values for:	And IMS supplies default values for:	Then the length (in hexadecimal) is:	Destination, Identification, SNAP options		16	Destination, Identification	SNAP options	12	Destination	Identification, SNAP options	10		Destination, Identification, SNAP options	2
If you supply values for:	And IMS supplies default values for:	Then the length (in hexadecimal) is:															
Destination, Identification, SNAP options		16															
Destination, Identification	SNAP options	12															
Destination	Identification, SNAP options	10															
	Destination, Identification, SNAP options	2															
3-10		<p>This 8-byte field tells IMS where to send SNAP output. You can direct output to the IMS log by specifying one of the following values: LOGbbbb</p> <hr/> <p>Directs the output to the IMS log. This is the default destination.</p> <hr/> <p><i>dcbaddr</i> Directs the output to the data set defined by this DCB address.</p> <p>The application program must open the data set before the SNAP call refers to it. This option is valid only in a batch environment. The output data set must conform to the rules for a z/OS SNAP data set.</p> <hr/> <p><i>ddname</i> Directs the output to the data set defined by the corresponding DD statement. The DD statement must conform to the rules for a z/OS SNAP data set. The data set specified by <i>ddname</i> is opened and closed for this SNAP request.</p> <p>In a DB/DC environment, you must supply the DD statement in the JCL for the control region.</p> <p>If the destination is invalid, IMS directs output to the IMS log.</p>															
11-18	cccccccc	<p>This is an eight-character name you can supply to identify the SNAP. If you do not supply a name, IMS uses the default value, NOTGIVEN.</p>															
19-22	cccc	<p>This four-character field identifies which data elements you want the SNAP output to include. YYYYN is the default.</p>															
19		<p><b>Buffer Pool:</b></p> <hr/> <p>Y Dump all buffer pools and sequential buffering control blocks with a SNAP call.</p> <hr/> <p>N Do not dump buffer pools or sequential buffering control blocks with a SNAP call.</p>															

Table 37. SNAP Operation Parameters (continued)

Byte	Value	Meaning
20		<b>Control Blocks:</b>
	Y	Dump control blocks related to the current DB PCB with a SNAP call.
	N	Do not dump control blocks related to the current DB PCB with a SNAP call.
21	Y	Dump all control blocks for this PSB with a SNAP call. Specifying Y in byte 21 produces a snap dump for the current DB PCB request in byte 20 to Y, regardless of the current value.
	N	Do not dump all control blocks for this PSB with a SNAP call. In this case, the current DB PCB SNAP request in position 20 is used as specified.
19-21	ALL	This is equivalent to specifying YYY in positions 19-21.
22		<b>Region:</b>
	Y	Dump the entire region on the DCB address or data set ddname that you supplied in bytes 3-10 with a SNAP call. IMS processes this request before it acts on any SNAP requests made in bytes 19-21. If the destination is the IMS log, IMS does not dump the entire region. Instead, it processes the request as if you had specified ALL.
	N	Do not dump the entire region with a SNAP call.
	S	Dump subpools 0-127 with a SNAP call.

After the SNAP call, IMS can return the AB, AD, or bb (blank) status code. For a description of these codes and the response required, see *IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS*.

## Usage

Any application program can issue this call.

## Restrictions

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

## STAT Call

This section contains product-sensitive programming interface information.

The Statistics (STAT) call is used in a CICS, IMS online, or batch program to obtain database statistics that might be useful for performance monitoring.

## Format

```

▶▶—STAT—┌──db pcb──┐──i/o area──stat function──▶▶
          └──aib──┘

```

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
STAT	X	X		X	



## Parameters

### *db pcb*

Specifies the DB PCB used to pass status information to the application program. The VSAM statistics used by the data sets associated with this PCB are not related to the type of statistics that is returned from the STAT call. This PCB must reference a full-function database. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the name of a full-function DB PCB.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list.

### *i/o area*

Specifies an area in the application program that is large enough to hold the requested statistics. This parameter is an output parameter. In PL/I, this parameter should be specified as a pointer to a major structure, array, or character string.

### *stat function*

Specifies a 9-byte area whose content describes the type and format of the statistics required. The first 4 bytes define the type of statistics requested and byte 5 defines the format to be provided. The remaining 4 bytes contain EBCDIC blanks. If the *stat function* that is provided is not one of the defined functions, an AC status code is returned. This parameter is an input parameter. The 9-byte field contains the following information:

- 4 bytes that define the type of statistics you want:

**DBAS** OSAM database buffer pool statistics

**DBES** OSAM database buffer pool statistics, enhanced or extended

**VBAS** VSAM database subpool statistics

**VBES** VSAM database subpool statistics, enhanced or extended

- 1 byte that gives the format of the statistics:

**F** Full statistics to be formatted. If you specify F, your I/O area must be at least 360 bytes for DBAS or VBAS and 600 bytes for DBES or VBES.

**O** Full OSAM database subpool statistics in a formatted form. If you specify O, your I/O area must be at least 360 bytes.

**S** Summary of the statistics to be formatted. If you specify S, your I/O area must be at least 120 bytes for DBAS or VBAS and 360 bytes for DBES or VBES.

- U** Full statistics to be unformatted. If you specify U, your I/O area must be at least 72 bytes.
- 4 bytes of EBCDIC blanks for normal or enhanced STAT call, or bE1b, for extended STAT call.

**Restriction:** The extended format parameter is supported by the DBESO, DBESU, and DBESF functions only.

Extended OSAM buffer pool statistics can be retrieved by including the 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 and SYNC read counts.

## Usage

The STAT call can be helpful in debugging because it retrieves IMS database statistics. It is also helpful in monitoring and tuning for performance. The STAT call retrieves OSAM database buffer pool statistics and VSAM database buffer supports.

When you request VSAM statistics, each issued STAT call retrieves the statistics for a subpool. 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 in ascending order based on buffer size. Statistics for index subpools always follow those for data subpools if any index subpool exists in the shared resource pool. The index subpools are also retrieved in ascending order based on buffer size.

For more information on the STAT call, see *IMS Version 9: Application Programming: Design Guide*.

## Restrictions

For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

---

## SYNC Call

The Synchronization Point (SYNC) call is used to release resources that IMS has locked for the application program.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
SYNC	X	X	X		

## Parameters

*i/o pcb*

Specifies the IO PCB for the call. This parameter is an input and output parameter.

*aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

**AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

**AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

**AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

## Usage

SYNC commits the changes your program has made to the database, and establishes places in your program from which you can restart, if your program terminates abnormally.

## Restrictions

The SYNC call is valid only in non-message driven BMPs; you cannot issue a SYNC call from an CPI-C driven application program.

For important considerations about using the SYNC call, see *IMS Version 9: Administration Guide: Database Manager*.

---

## TERM Call (CICS Online Programs Only)

The Terminate (TERM) call is used to terminate a PSB in a CICS online program.

The ODBA interface does not support this call.

## Format

▶▶—TERM—◀◀

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
TERM	X	X			

## Usage

If your program needs to use more than one PSB, you must issue a TERM call to release the first PSB it uses and then issue a second PCB call to schedule the second PSB. The TERM call also commits database changes.

The only parameter in the TERM call is the call function: TERM or Tbbb. When your program issues the call, CICS terminates the scheduled PSB, causes a CICS sync point, commits changes, and frees resources for other tasks.

## Restrictions

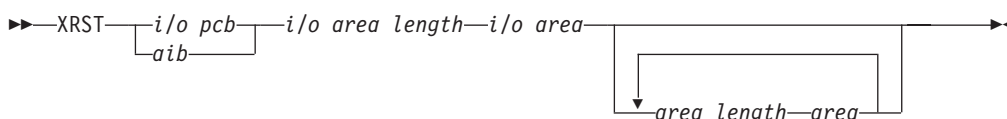
For function shipping in the CICS environment, the local and remote CICS must both be DBCTL.

## XRST Call

The Extended Restart (XRST) call is used to restart your program. If you use the symbolic Checkpoint call in your program, you **must** precede it with an XRST call that specifies checkpoint data of blanks.

The ODBA interface does not support this call.

## Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
XRST	X	X	X	X	X

## Parameters

### *i/o pcb*

Specifies the I/O PCB for the call. XRST must refer to the I/O PCB. This parameter is an input and output parameter.

### *aib*

Specifies the AIB for the call. This parameter is an input and output parameter. The following fields must be initialized in the AIB:

#### **AIBID**

Eye catcher. This 8-byte field must contain DFSAIBbb.

#### **AIBLEN**

AIB lengths. This field must contain the actual length of the AIB that the application program obtained.

#### **AIBRSNM1**

Resource name. This 8-byte, left-justified field must contain the PCB name, IOPCBbbb.

#### **AIBOALEN**

I/O area length. This field must contain the length of the I/O area specified in the call list. This parameter is not used during the XRST call. For compatibility reasons, this parameter must still be coded.

### *i/o area length*

This parameter is no longer used by IMS. For compatibility reasons, this parameter must still be included in the call, and it must contain a valid address. You can get a valid address by specifying the name of any area in your program.

*i/o area*

Specifies a 14-byte area in your program. This area must be either set to blanks if starting your program normally or, if performing an extended restart, have a checkpoint ID.

*area length*

Specifies a 4-byte field in your program that contains the length (in binary) of the area to restore. This parameter is an input parameter. You can specify up to seven area lengths. For each area length, you must specify the *area* parameter. All seven *area* parameters (and corresponding *area length* parameters) are optional. When you restart the program, IMS restores only the areas specified on the CHKP call.

The number of areas you specify on an XRST call must be less than or equal to the number of areas you specify on a CHKP call.

*area*

Specifies the area in your program that you want IMS to restore. You can specify up to seven areas. Each area specified must be preceded by an *area length*. This is an input parameter.

## Usage

Programs that wish to issue Symbolic Checkpoint calls (CHKP) must also issue the Extended Restart call (XRST). The XRST call must be issued only once and should be issued early in the execution of the program. It does not need to be the first call in the program. However, it must precede any CHKP call. Any Database calls issued before the XRST call are not within the scope of a restart.

To determine whether to perform a normal start or a restart, IMS evaluates the I/O area provided by the XRST call or CKPTID= value in the PARM field on the EXEC statement in your program's JCL.

### Starting Your Program Normally

When you are starting your program normally, the I/O area pointed to in the XRST call must contain blanks and the CKPTID= value in the PARM field must be nulls. This indicates to IMS that subsequent CHKP calls are symbolic checkpoints rather than basic checkpoints. Your program should test the I/O area after issuing the XRST call. IMS does not change the area when you are starting the program normally. However, an altered I/O area indicates that you are restarting your program. Consequently, your program must handle the specified data areas that were previously saved and that are now restored.

### Restarting Your Program

You can restart the program from a symbolic checkpoint taken during a previous execution of the program. The checkpoint used to perform the restart can be identified by entering the checkpoint ID either in the I/O area pointed to by the XRST call (left-most justified, with the rest of the area containing blanks) or by specifying the ID in the CKPTID= field of the PARM= parameter on the EXEC statement in your program's JCL. (If you supply both, IMS uses the CKPTID= value specified in the parm field of the EXEC statement.)

The ID specified can be:

- A 1 to 8-character extended checkpoint ID
- A 14-character "time stamp" ID from message DFS0540I, where:
  - IIII is the region ID
  - DDD is the day of the year

- HHMSST is the time in hours, minutes, seconds, and tenth of a second
- The 4-character constant "LAST". (BMPs only: this indicates to IMS that the last completed checkpoint issued by the BMP will be used for restarting the program)

The system message DFS0540I supplies the checkpoint ID and the time stamp.

The system message DFS682I supplies the checkpoint ID of the last completed checkpoint which can be used to restart a batch program or batch message processing program (BMP) that was abnormally terminated.

If the program being restarted is in either a batch region or a BMP region, and the checkpoint log records no longer reside on the Online Log Data Set (OLDS) or System Log Data Set (SLDS), the //IMSLOGR DD defining the log data set must be supplied in the JCL for the BATCH or BMP region. IMS reads these data sets and searches for the checkpoint records with the ID that was specified.

**Restriction:** To issue a checkpoint restart from a batch job, you must use the original job name, or IMS cannot locate the checkpoint and the job fails with a U0102.

At completion of the XRST call the I/O area always contains the 8-character checkpoint ID used for the restart. An exception exists when the checkpoint ID is equal to 8 blank characters; the I/O area then contains a 14-character time stamp (IIIIDDHHMSST).

Also check the status code in the I/O PCB. The only successful status code for an XRST call are blanks.

### Position in the Database after Issuing XRST

The XRST call attempts to reposition all databases to the position that was held when the last checkpoint was taken. This is done by including each PCB and PCB key feedback area in the checkpoint record. Issuing XRST causes the key feedback area from the PCB in the checkpoint record to be moved to the corresponding PCB in the PSB that is being restarted. Then IMS issues a GU call, qualified with the concatenated key (using the C command code), for each PCB that held a position when the checkpoint was taken.

After the XRST call, the PCB reflects the results of the GU repositioning call, not the value that was present when the checkpoint was taken. The GU call is not made if the PCB did not hold a position on a root or lower-level segment when the checkpoint was taken. A GE status code in the PCB means that the GU for the concatenated key was not fully satisfied. The segment name, segment level, and key feedback length in the PCB reflect the last level that was satisfied on the GU call. A GE status code can occur because IMS is unable to find a segment that satisfies the segment search argument that is associated with a Get call for one of the following reasons:

- The call preceding the checkpoint call was a DLET call issued against the same PCB. In this case, the position is correct because the not-found position is the same position that would exist following the DLET call.

**Restriction:** Avoid taking a checkpoint immediately after a DLET call.

- The segment was deleted by another application program between the time your program terminated abnormally and the time you restarted your program. A GN call issued after the restart returns the first segment that follows the deleted segment or segments.

The above explanation assumes that position at the time of checkpoint was on a segment with a unique key. XRST cannot reposition to a segment if that segment or any of its parents have a non unique key.

For a DEDB, the GC status code is received when position is not on a segment but at a unit-of-work (UOW) boundary. Because the XRST call attempts to reestablish position on the segment where the PCB was positioned when the symbolic checkpoint was taken, the XRST call does not reestablish position on a PCB if the symbolic checkpoint is taken when the PCB contains a GC status code.

If your program accesses GSAM databases, the XRST call also repositions these databases. For more information on processing GSAM databases, see Chapter 10, "Processing GSAM Databases," on page 219.

## Restrictions

If your program is being started normally, the first 5 bytes of the I/O area must be set to blanks.

If your program is restarted and the CKPTID= value in the PARM field of the EXEC statement is not used, then the right-most bytes beyond the checkpoint ID being used in the I/O area must be set to blanks.

The XRST call is allowed only from Batch and BMP application programs.





---

## Chapter 6. Monitoring Your Position in the Database

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.

### **In this Chapter:**

- “Understanding Current Position in the Database”
- “Current Position after Unsuccessful Calls” on page 194

---

### Understanding Current Position in the Database

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. “Current Position after Unsuccessful Calls” on page 194 explains current position in the database after an unsuccessful 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 CHPK 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 SSAs 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 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.

**Example:** Suppose you issue the following call against the data structure shown in Figure 34 on page 190.

```
GU  ABBBBBBB(AKEYBBBB=bA1)
    BBBBBBBB(BKEYBBBB=bB11)
    DBBBBBBB(DKEYBBBB=bD111)
```

The structure in the figure contains six segment types: A, B, C, D, E, and F. Figure 34 on page 190 shows one database record, the root of which is A1.

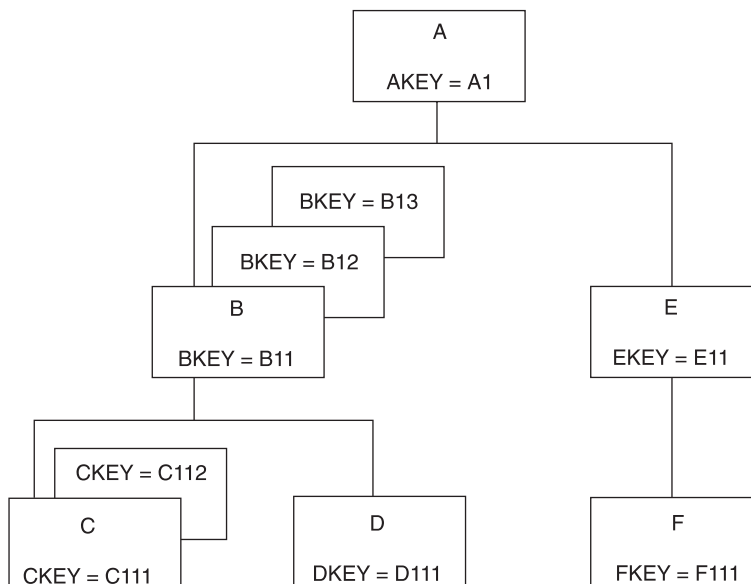


Figure 34. Current Position Hierarchy

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 SSAs. 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 F Command Code” on page 31.

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 SSAs that it cannot find the segment you have requested (GE status code). “Current Position after Unsuccessful Calls” on page 194 explains where your position is when you receive a 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.

## Position after Retrieval Calls

After you issue any kind of successful retrieval call, position is immediately after 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.

**Example:** If you issue the following call against the database shown in Figure 34 on page 190, 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(AKEYBBBB=bA1)
     BBBBBBBB(BKEYBBBB=bB11)
     CBBBBBBB(CKEYBBBB=bC111)
```

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 that was shown above but include the D command code in the SSAs for segments A and B, 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*D(AKEYBBBB=bA1)
     BBBBBBBB*D(BKEYBBBB=bB11)
     CBBBBBBB(CKEYBBBB=bC111)
```

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 is immediately after the segment occurrence you deleted. This is true when you delete a segment occurrence with or without dependents.

**Example:** If you issue the call shown Figure 35 to delete segment C111, current position is immediately after segment C111. Then, if you issue an unqualified GN call, IMS returns segment C112.

```
GHU  ABBBBBBB(AKEYBBBB=bA1)
      BBBBBBBB(BKEYBBBB=bB11)
      CBBBBBBB(CKEYBBBB=bC111)
DLET
```

*Figure 35. Example Code: Deleting Segment C11*

Figure 36 on page 192 shows what the hierarchy looks like after this call. The successful DLETcall has deleted segment C111.

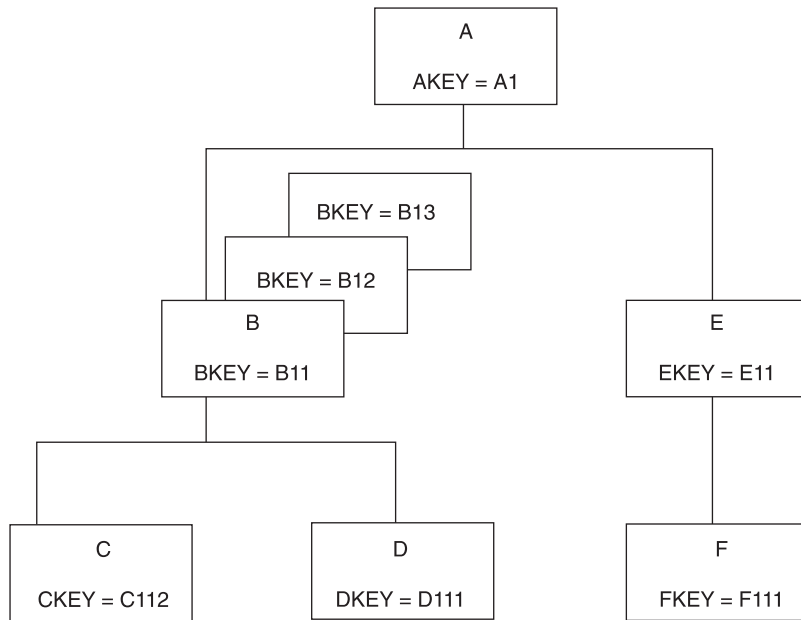


Figure 36. Hierarchy after Deleting a Segment

When you issue a successful DLET call for a segment occurrence that has dependents, IMS deletes the dependents, and the segment occurrence. Current position is still immediately after the segment occurrence you deleted. An unqualified GN call returns the segment occurrence that followed the segment you deleted.

**Example:** If you delete segment B11 in the hierarchy shown in Figure 36, IMS deletes its dependent segments, C112 and D111, as well. Current position is immediately after segment B11, just before segment B12. If you then issue an unqualified GN call, IMS returns segment B12. Figure 37 shows what the hierarchy looks like after you issued this call.

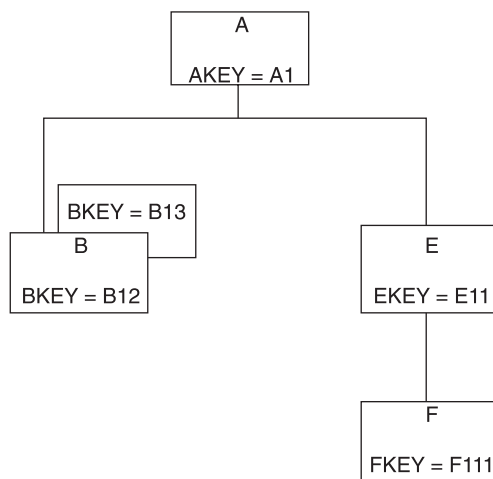


Figure 37. Hierarchy after Deleting a Segment and Dependents

Because IMS deletes the segment's dependents, you can think of current position as being immediately after the last (lowest, right-most) dependent. In the example in Figure 36, this is immediately after 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. See “Current Position after Unsuccessful Calls” on page 194 regarding position after unsuccessful calls.

## 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 is immediately after the lowest segment that is retrieved by the Get Hold call that you issued before the REPL call.

**Example:** If you retrieve segment B13 in Figure 37 on page 192 using a GHU instead of a GU call, change the segment in the I/O area, and then issue a REPL call, current position is immediately after segment B13.

## Position after ISRT

After you add a new segment occurrence to the database, current position is immediately after the new segment occurrence.

**Example:** In Figure 38 on page 194, if you issue the following call to add segment C113 to the database, current position is immediately following segment C113. An unqualified GN call would retrieve segment D111.

```
ISRT  Abbbbbbb(AKEYbbbb=bA1)
      Bbbbbbbb(BKEYbbbb=bB11)
      Cbbbbbbb
```

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. “ISRT Call” on page 138 explains these rules.

If you insert several segment occurrences using the D command code, current position is immediately after the lowest segment occurrence that is inserted.

**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. Figure 38 on page 194 shows what the hierarchy looks like after you insert these segment occurrences. The call to do this looks like this:

```
ISRT  Abbbbbbb(AKEYbbbb=bA1)
      Bbbbbbbb*D
      Cbbbbbbb
```

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 is immediately after 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.

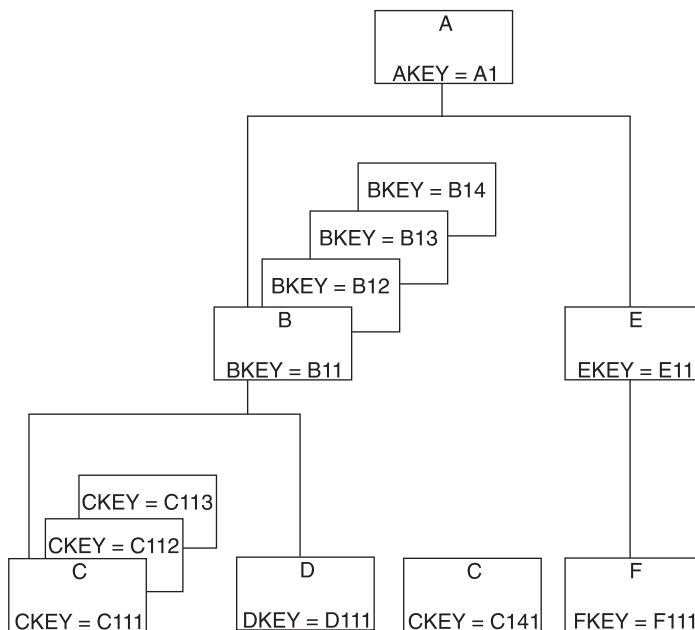


Figure 38. Hierarchy after Adding New Segments and Dependents

## 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. You need to know how IMS establishes this position to understand the U and V command codes described in “Command Codes” on page 28. 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.

In “Understanding Current Position in the Database” on page 189 you saw what current position is, why and when it is important, and how successful DL/I calls affect it. But chances are that not every DL/I call that your program issues will be completely successful. When a call is unsuccessful, you should understand how to determine where your position in the database is after that 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.

**Example:** In Figure 39 on page 195, if you had just successfully issued the GU call with the SSAs shown below, IMS has a position established at each hierarchic level.

GU Abbbbbbb (AKEYbbbb=bA1)  
 Bbbbbbbb (BKEYbbbb=bB11)  
 Cbbbbbbb (CKEYbbbb=bC111)

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

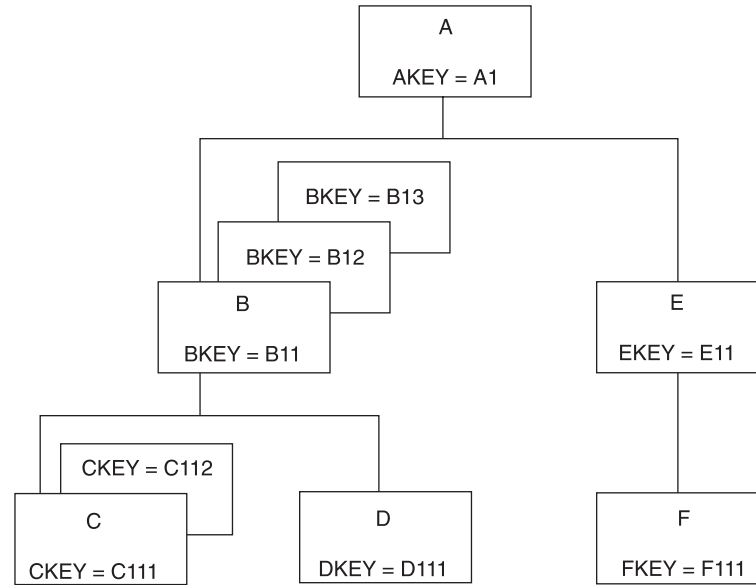


Figure 39. DL/I Positions

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 SSAs 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. The way that you code the SSAs in the call determines whether IMS can move forward and try again under another parent.

**Example:** Suppose you issue the following GN call to retrieve the C segment with the key of C113 in the hierarchy shown in Figure 39 on page 195.

```
GN  Abbbbbbb(AKEYbbbb=bA1)
    Bbbbbbbb(BKEYbbbb=bB11)
    Cbbbbbbb(CKEYbbbb=bC113)
```

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 SSAs 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 is immediately after 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.

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 is immediately after segment B11. An unqualified GN call would return B12.

If A and B both have non unique keys, current position after the previous call is immediately after 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  Abbbbbbb(AKEYbbbb=bA1)
    Bbbbbbbb(BKEYbbbb>=B11)
    Cbbbbbbb(CKEYbbbb=bC113)
```

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 IMS's search for segment C113 to only one B segment, because you have used the greater-than-or-equal-to operator. IMS's 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; IMS's current position is immediately following 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 IMS's 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  ABBBBBBB(AKEYBBBB=bA1)
    BBBBBBBB(BKEYBBBB>=B11*BDATABBB=b14)
    CBBBBBBB(CKEYBBBB=bC113)
```

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, IMS's current position is immediately after 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 IMS's 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, IMS's 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.

**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  Abbbbbbb*U  
    Bbbbbbb*U  
    Cbbbbbb
```

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.

---

## Chapter 7. 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.

### In this Chapter:

- “Overview of Multiple Qualification Statements”
- “Example using Multiple Qualification Statements” on page 200
- “Multiple Qualification Statements for HDAM, PHDAM, or DEDB” on page 201

---

### Overview of Multiple Qualification Statements

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. “Multiple Qualification Statements with Secondary Indexes” on page 212 describes its use.

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:

```
ROOTKEYb=b10&FIELDDBbb=XYZ+ROOTKEYbb=10&FIELDDBbb=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 FIELDDB must be equal to either ABC or XYZ.

```
ROOTKEYb=>10&ROOTKEYb=<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.

```
ROOTKEYb=>10&ROOTKEYb=<20+ROOTKEYb=>110&ROOTKEYb=<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. See “How Logical Relationships Affect Your Programming” on page 216 for more information about these considerations.

---

## Example using Multiple Qualification Statements

The easiest way to understand multiple qualification statements is to look at an example:

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(PATNObbbEQ04120)
   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 after or equal to January 1, 1992, and before or equal to 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(PATNObbbEQ05682)
    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, PHDAM, or 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).

**Recommendation:** When the organization is HDAM or DEDB, do not use calls that allow a range of values at the root level.

For more details about HDAM or PHDAM databases, see *IMS Version 9: Administration Guide: Database Manager*.



## Chapter 8. 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 Figure 40, 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.

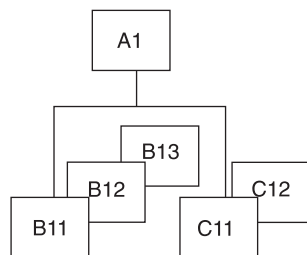


Figure 40. Multiple Processing

### **In this Chapter:**

- “Multiple Positioning”
- “Advantages of Using Multiple Positioning” on page 206
- “Using Multiple DB PCBs” on page 208

## 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. This section explains what multiple positioning is, why it is useful, and how it affects your programming.

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.

### **Definitions:**

- *Single positioning* means that 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* means that 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.

**Example:** Suppose you issue these two calls using the hierarchy shown in Figure 41:

```

GU  Abbbbbbb(AKEYbbbb=bA1)
    Bbbbbbbb(BKEYbbbb=bB11)
    Cbbbbbbb(CKEYbbbb=bC111)

GN  Ebbbbbbb(EKEYbbbb=bE11)

```

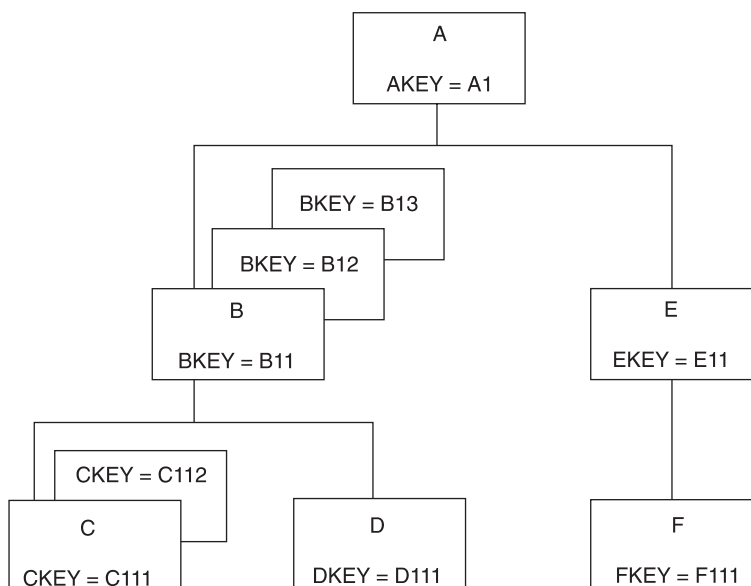


Figure 41. Multiple Positioning Hierarchy

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.



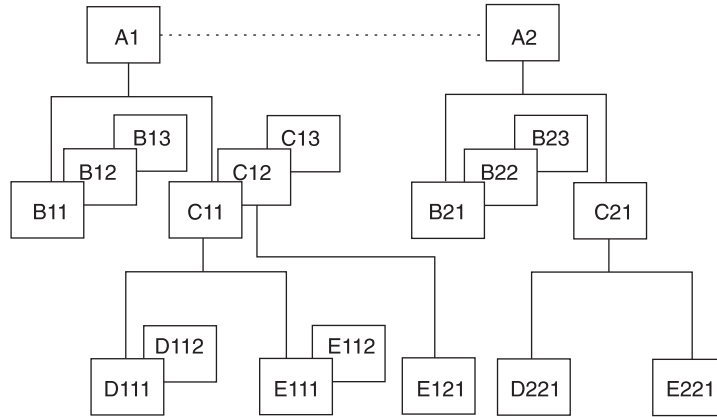


Figure 42. Single and Multiple Positioning Hierarchy

The examples that follow compare the results of single and multiple positioning using the hierarchy in Figure 42.

Table 38. Results of Single and Multiple Positioning with DL/I Calls

Sequence	Result of Single Positioning	Result of Multiple Positioning
<b>Example 1</b>		
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	Not found
GNP C	C13	C13
GNP B	Not found	Not found
GNP C	Not found	Not found
<b>Example 2</b>		
GU A (where AKEY equals A1)	A1	A1
GN B	B11	B11
GN C	C11	C11
GN B	B21	B12
GN C	C21	C12
<b>Example 3</b>		
GU A (where AKEY equals A1)	A1	A1
GN C	C11	C11
GN B	B21	B11
GN B	B22	B12
GN C	C21	C12
<b>Example 4</b>		
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 are:

- 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.

### Using 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 SSAs 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.

### Using 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.

**Using 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 SSAs.

**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, after issuing an unqualified call, you issue a qualified call for a segment in a different hierarchic path, 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=bA2)
```

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=bA1)
```

Issuing this call as a path call does not cancel position.

---

## Using 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.

**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. On the other hand, 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 9. Secondary Indexing and Logical Relationships

This chapter describes two ways in which IMS can provide flexibility in how your program views the data. 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.

### **In this Chapter:**

- "How Secondary Indexing Affects Your Program"
- "Processing Segments in Logical Relationships" on page 214

---

### 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. This section explains how secondary indexing affects your programming.

For more information about secondary indexes and examples, see *IMS Version 9: Application Programming: Design Guide*.

### 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 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.

**Related Reading:** For more detailed information about generating a DBD and a PSB, refer to the *IMS Version 9: Utilities Reference: System*.

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.

**Example:** For you 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 9: Utilities Reference: System*.

### 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.

**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. Figure 43 on page 213 shows the three secondary indexing segments.



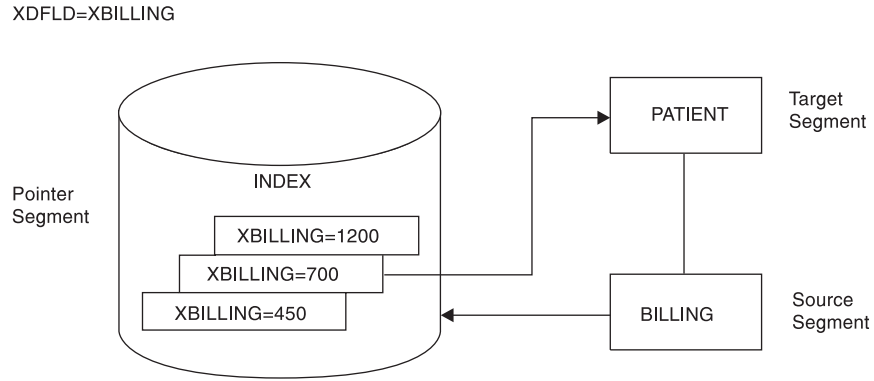


Figure 43. Example of Using the Dependent AND

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**

**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. Figure 44 shows the four secondary indexing segments.

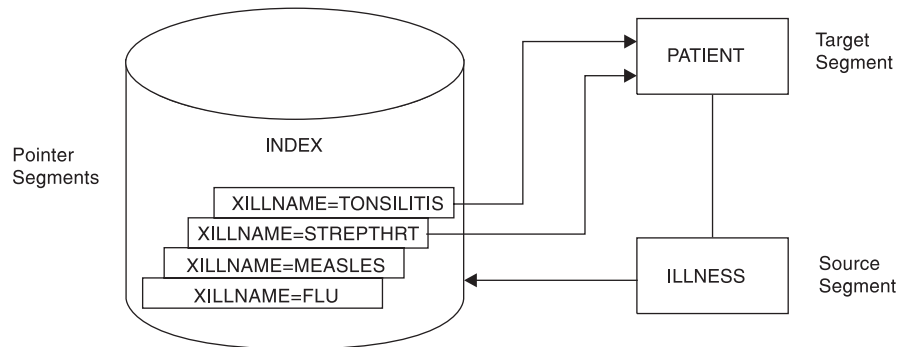


Figure 44. Example of Using the Independent AND

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 PATIENTb(XILLNAME=TONSILITIS#XILLNAME=bSTREPTHRT)
```

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.

## What DL/I Returns with a Secondary Index

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).

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 *IMS Version 9: Application Programming: Design Guide* gives some examples of this.

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 9: Messages and Codes, Volume 1* 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. This section uses the example about an inventory application program that processes data in a purchasing database, but which also needs access to a segment in a patient database.

**Related Reading:**

- For more information about application programming requirements that logical relationships can satisfy, see *IMS Version 9: Application Programming: Design Guide*.
- For a full description of the inventory application program example, see *IMS Version 9: Application Programming: Design Guide*.

**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 used throughout this section 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. Figure 45 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.)

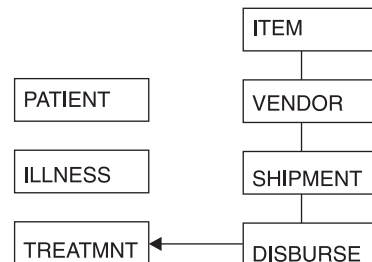


Figure 45. Patient and Item Hierarchies

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:

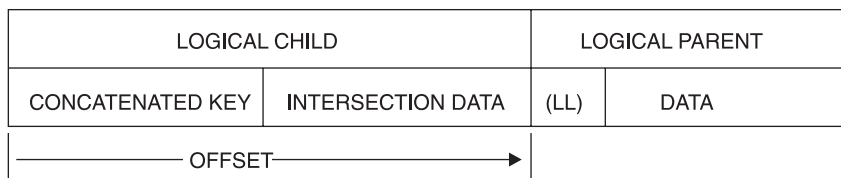


Figure 46. Concatenated Segment

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 in the following ways: 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, you should have the following information from the DBA (or the person at your installation responsible for database design):

- 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

Inserting a logical child segment has the following requirements:

- 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, as illustrated in Figure 46 on page 216.

## Status Codes for Logical Relationships

The following 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 10. Processing GSAM Databases

GSAM databases are available to application programs that can run as batch programs, batch-oriented BMPs, or transaction-oriented BMPs. If your program accesses GSAM databases, you need to consider the following points as you design your program:

- 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 it like a sequential nonhierarchic database. The MVS 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 nonkeyed, 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 nonhierarchic database, it has no segments, no keys, and no parentage.

### In this Chapter:

- “Accessing GSAM Databases”
- “GSAM Record Formats” on page 222
- “GSAM I/O Areas” on page 223
- “GSAM Status Codes” on page 223
- “Symbolic CHKP and XRST with GSAM” on page 224
- “GSAM Coding Considerations” on page 224
- “Origin of GSAM Data Set Characteristics” on page 225

---

## Accessing GSAM Databases

The calls you use to access 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. Programs 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.

Calls to GSAM databases can use either the AIBTDLI or the PCB interface. For information on the AIBTDLI interface, see “The AIBTDLI Interface” on page 111.

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. Table 39 on page 220 shows the order and lengths of these fields. Because GSAM is not a hierarchic 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), and the eighth field (number of sensitive segments). Even though GSAM does not use these fields, you need to define them in the order and length shown in Table 39 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. Consider the following points about the entry statement:

- When you code the entry statement in COBOL, Pascal, C, and assembler language programs, it must list the names of the DB PCB masks in your program.
- When you code the entry statement in PL/I programs, it 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 39. GSAM DB PCB Mask

Descriptor	Byte Length	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
Database name <sup>1</sup>	8	X	X	X	X	X
Segment level number <sup>2</sup>	2	N/A	N/A	N/A	N/A	N/A
Status code <sup>3</sup>	2	X	X	X	X	X
Processing options <sup>4</sup>	4	X	X	X	X	X
Reserved for IMS <sup>5</sup>	4	X	X	X	X	X
Segment name <sup>6</sup>	8	N/A	N/A	N/A	N/A	N/A
Length of key feedback area and undefined-length records area <sup>7</sup>	4	X	X	X	X	X
Number of sensitive segments <sup>8</sup>	4	N/A	N/A	N/A	N/A	N/A
Key feedback area <sup>9</sup>	8	X	X	X	X	X
Length of undefined-length records <sup>10</sup>	4	X	X	X	X	X



**Notes:**

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. This is the sum of the lengths of the key feedback and the undefined-length record areas described below.
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.
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.

## Retrieving and Inserting GSAM Records

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 means that you have reached the end of the database. GSAM automatically closes the database when you reach the end of it. To create a new data set or to add new records to the end of 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, but you must supply the record's address. To do this, use a record search argument (RSA). An RSA is similar to an 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

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 in the TTR track record) format.

Before you can give GSAM the RSA, you must know the RSA yourself. To do this, you must know in advance what records you want to retrieve at a later time. When you are retrieving records sequentially or adding records to the end of the GSAM database, you can include a parameter on the GN or ISRT call that tells GSAM to return the address of that record to a certain area in your program, as shown in Table 40 on page 225. Save this address until you want to retrieve that particular record. At that time, you issue a GU call for the record and give the address of its RSA as a parameter of the GU call. GSAM returns the record to the I/O area that you named as one of the call parameters. Do this on DASD only. When using buffered I/O, this may degrade performance for output PCBs.

You can also use a GU call and an RSA to position yourself at a certain place in the GSAM database. If you place a doubleword, consisting of a fullword containing the binary value "1" followed by a fullword containing the binary value "0", in the RSA and issue a GU call using that RSA, GSAM repositions you to the first record in the database.

## Explicitly Opening and Closing a GSAM Database

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 in the following two situations:

- 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.
- If 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 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. This is the tenth field in Table 39 on page 220. 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 MVS 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. "Origin of GSAM Data Set Characteristics" on page 225 explains what you can use to override each type of record format.

---

## 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, after checking the status codes, you find that you have an error and terminate your program, 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.

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.

---

## GSAM Coding Considerations

The calls your program uses to access GSAM databases are not the same as the DL/I calls. This section tells how to code GSAM calls and GSAM data areas. The system service calls that you can use with GSAM are symbolic CHKP and XRST.

Table 40 summarizes GSAM database calls. The five calls you can use to process GSAM databases are: CLSE, GN, GU, ISRT, and OPEN. The COBOL, PL/I, Pascal, C, and assembler call formats and parameters for these calls are the same and are described in Table 40 on page 225. GSAM calls do not differ significantly from DL/I calls, but GSAM calls must reference the GSAM PCB, and they do not use SSAs.

Table 40. Summary of GSAM Calls

Call Formats	Meaning	Use	Options	Parameters
CLSE	Close	Explicitly closes GSAM database	None	function, gsam pcb
GNbb	Get Next	Retrieves next sequential record	Can supply address for RSA to be returned	function, gsam pcb, i/o area [,rsa name]
GUbb	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]

## 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

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.

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
- 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.

## 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:

- For input data sets, use DISP=OLD.
- For output data sets, you have a number of options:
  - If you are creating an output data set allocated by the DD statement, use DISP=NEW.
  - To create an output data set that was previously cataloged, but is now empty, use DISP=MOD.
  - When restarting the step, use DISP=OLD.
  - Finally, to add new records to the end of an existing data set, use DISP=MOD.

## Using Extended Checkpoint Restart for GSAM Data Sets

**Recommendation:** If you are using extended checkpoint restart for GSAM data sets:

- 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.
- Additionally, keep in mind the following points:
  - 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, but, in all other respects, the checkpoint has completed normally.
  - 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 and checkpoint time.
  - GSAM uses the relative track and record (TTR) on the volume to position GSAM DASD data sets when restarting. For a tape data set, the relative record on the volume is used. If a data set is copied between checkpoint and restart, the TTR on the volume for DASD or the relative record on the volume cannot be changed. To avoid this problem, do the following:



1. Copy the data set to the same device type.
2. Use RECFM=U for both the input and the output data set to avoid any reblocking.
3. Be sure that each copied volume contains the same number of records as the original volumes when copying a multivolume data set.

## Use of Concatenated Data Sets 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.

## Suggested Method for Specifying GSAM Data Set Attributes

**Recommendation:** When specifying GSAM data set attributes:

- On the DBD, specify RECFM. (It is required.)
- On the DATASET statement, specify the logical record length using RECORD=.
- 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 adds 2 bytes. The record size of the GSAM database is 2 bytes greater than the longest segment that is passed to IMS by the application program.

## DLI or DBB Region Types and GSAM

The two kinds of batch regions are the DLI batch region and the DBB batch region. The only difference between them is the source for DLI control blocks. For a DLI region, the source for control blocks is PSBLIB and DBDLIB. For a DBB region, the source for control blocks is the ACBLIB, as identified by the //IMSACB DD statement. When you initialize a DLI, DBB or BMP region using GSAM, you must include an //IMS DD statement and GSAM database DD statements. Note that when DBB or BMP regions are not using GSAM, no //IMS DD statement is required. The //IMS DD statements are required for loading PSBs and DBDs, and for building GSAM control blocks. GSAM does not obtain PSB and DBD information from ACBLIB.

```

//STEP      EXEC      PGM=DFSRR00,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)
           .
           .
           .
/*

```

Figure 47. //IMS DD Statement Example



## Chapter 11. Processing Fast Path Databases

This chapter contains information on Fast Path database calls and MSDB and DEBD information that is required by Fast Path calls. Fast Path message calls appear in *IMS Version 9: Application Programming: Transaction Manager* manual.

**Restriction:** This DEBD information applies to CICS users with DBCTL. MSDBs and cannot be accessed through CICS and DBCTL.

The two kinds of Fast Path databases are:

- Main storage databases (MSDBs) 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.

### In this Chapter:

- “Fast Path Database Calls”
- “MSDBs and DEDBs: Overview” on page 230
- “Processing MSDBs and DEDBs” on page 231
- “Restrictions on Using Calls for MSDBs” on page 237
- “Processing DEDBs (IMS, CICS with DBCTL)” on page 238
- “Restrictions on Using Calls for DEDBs” on page 246
- “Fast Path Coding Considerations” on page 247

**Related Reading:** For more information on the types of processing requirements the two types of Fast Path databases satisfy, see *IMS Version 9: Administration Guide: Database Manager*. This section contains information on how to write programs to access data in MSDBs and DEDBs.

## Fast Path Database Calls

Table 41 summarizes the database calls you can use with Fast Path databases.

Table 41. Summary of Fast Path Database Calls

Function Code	Types of MSDBs:			DEDBs
	Nonterminal-Related	Terminal-Related Fixed	Terminal-Related Dynamic	
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

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 above 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. Table 42 shows which calls you can use with these command codes.

Table 42. 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

## MSDBs and DEDBs: Overview

This section briefly introduces the two Fast Path database types: MSDBs and DEDBs.

### 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.

#### Types of MSDBs

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. One way to understand the

differences between these types of databases and why you would use each one, is to look at the examples of each in “Bank Account Example” on page 22.

## 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 9: Administration Guide: System*.

---

## Processing MSDBs and DEDBs

This section describes update calls, commit point processing, and data locking for MSDBs and DEDBs.

## Updating Segments in an MSDB or DEDB: 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 Table 10 on page 23 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 a Field's Contents: 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. Table 43 shows the structure and format of an FSA.

Table 43. FSA Structure

FLD NAME	SC	OP	FLD VALUE	CON
8	1	1	Variable	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 not, you receive one of the status codes listed below. If IMS returns a nonblank status code in the FSA, it returns an FE status code to the PCB to indicate this. 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.
- 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 a Field's Contents: 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.

Using Table 43 on page 232 FLD/CHANGE works like this:

- 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.

### An Example of Using FLD/VERIFY and FLD/CHANGE

The example in this section uses the bank account segment shown in Table 10 on page 23. 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 following list describes each of the three FSAs:

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:

```
BALANCEbbH10000*
```

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

This section describes the MSDB commit view and DEDBs with an MSDB commit view. (The following explanation assumes that you are already familiar with the concepts of commit point processing, as described in *IMS Version 9: Application Programming: Design Guide*.)

### 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, on the other hand, 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

Your existing application programs can use either the MSDB commit view or the default DEDB 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 9: Messages and Codes, Volume 1* for information about that status code.

Figure 48 shows an example of a PCB that specifies the VIEW option.

```

PCB      ,                               *00000100
        TYPE=DB,                          *00000200
        NAME=DEDBJN21,                    *00000300
        PROCOPT=A,                        *00000400
        KEYLEN=30,                        *00000500
        VIEW=MSDB,                         *00000600
        POS=M                              00000700

```

Figure 48. Sample PCB Specifying View=MSDB

## 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.



**Related Reading:** For more information on HSSP, see *IMS Version 9: Administration Guide: Database Manager*.

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. 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.

---

## Restrictions on Using Calls for MSDBs

To retrieve segments from an MSDB <sup>1</sup>, 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 keep in mind 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).

---

1. This section does not apply to CICS users.

- 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 9: Administration Guide: Transaction Manager* for more information on ETO and LU 6.2.
- MSDBs cannot be shared among IMS subsystems in a sysplex group. When using the Fastpath 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).

---

## Processing DEDBs (IMS, CICS with DBCTL)

This section explains subset pointers, the POS call, data locking, and the P and H processing options.

### Processing DEDBs with Subset Pointers

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. (This is described later in “Using Subset Pointers”.) Each subset pointer points to the start of a new subset. For example, in Figure 49, 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.

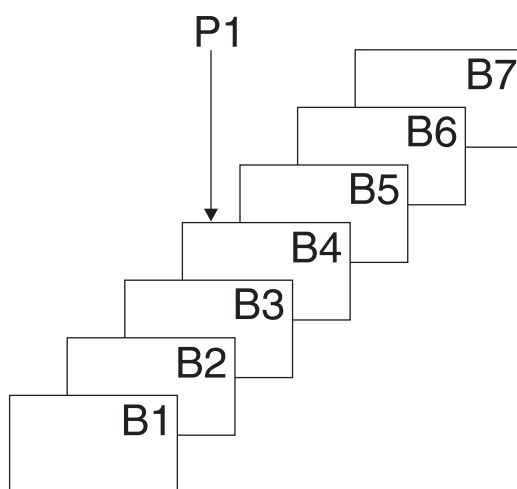


Figure 49. Processing a Long Chain of Segment Occurrences with Subset Pointers

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.

Figure 50 and Figure 51 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 Figure 50.

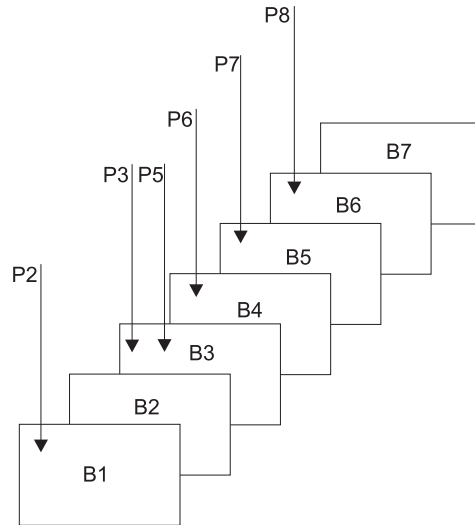


Figure 50. Examples of Setting Subset Pointers

You can also define a one-to-one relationship between the pointers and the segments, as shown in Figure 51.

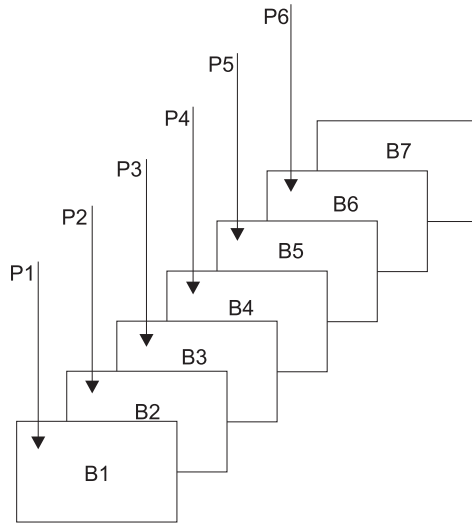


Figure 51. Additional Examples of Setting Subset Pointers

Figure 52 on page 240 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.

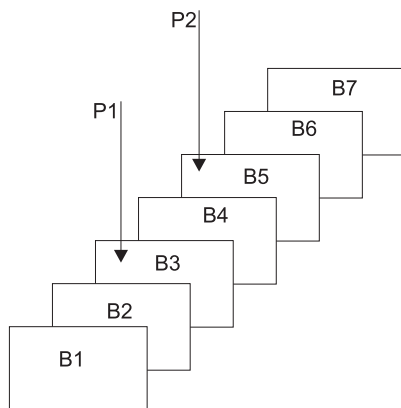


Figure 52. How Subset Pointers Divide a Chain into Subsets

### 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.

### Using Subset Pointers

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 "Command Codes" on page 28, 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 Table 44.

Table 44. Unqualified SSA with Subset Pointer Command Code

Seg Name	*	Cmd Code	Ssptr.	b
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 Table 45.

Table 45. 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

The examples in this section use calls with unqualified SSAs. The examples are based on Sample Application Program, which is described in “Fast Path Coding Considerations” on page 247.

**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.

**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  Abbbbbbb(Akeybbbb=bA1)
      Bbbbbbbb*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  Abbbbbbb(Akeybbbb=bA1)
      Bbbbbbbb*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.

### 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.

## Retrieving Location with the POS Call (for DEDB Only)

With the POS (Position) call, you can:

- 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.
- Retrieve the time stamp of a sequential dependent or Logical Begin.
- 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.

"POS Call" on page 142 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 Table 46. This LL field is described after Table 46.

Table 46. Qualified POS Call: Keywords and Map of I/O Area Returned

	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.



<b>Position</b>	An 8-byte field with binary zeros.
<b>Unused SDEP CIs</b>	A 4-byte field with binary zeros.
<b>Unused IOV CIs</b>	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.

If you want a DEDB to have an MSDB commit view, refer to “Commit-Point Processing in MSDBs and DEDBs” on page 235.

## Crossing a UOW Boundary (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 9: Administration Guide: Database Manager*.

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.

## Crossing the UOW Boundary (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

```

## Data Locking

For information on how data locking is handled for DEDBs, see “Data Locking for MSDBs and DEDBs” on page 236.

---

## Restrictions on Using Calls for DEDBs

This section provides information on which calls you can use with 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.

### **Exception:**

- 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. (See “Crossing a UOW Boundary (P Processing Option)” on page 245.)

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. Subset pointers are explained in “Processing DEDBs with Subset Pointers” on page 238.

## 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 9: Utilities Reference: Database and Transaction Manager*.

## 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.

You can use the following calls to process MSDBs:

- 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 the following calls to process a DEDB:

- DEQ
- DLET
- FLD
- GU and GHU
- GN and GHN
- GNP and GHNP
- ISRT
- POS

- REPL

---

## Chapter 12. Recovering Databases and Maintaining Database Integrity

This chapter describes the programming tasks of issuing checkpoints, restarting programs, and maintaining database integrity.

### **In this Chapter:**

- “Issuing Checkpoints”
- “Restarting Your Program and Checking for Position”
- “Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)” on page 250
- “Reserving Segments for the Exclusive Use of Your Program” on page 256

---

### 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 9: Application Programming: Design Guide* 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. For more information about the differences, see “Commit-Point Processing in MSDBs and DEDBs” on page 235.

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 following 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 “CHKP (Basic) Call” on page 150 and “CHKP (Symbolic) Call” on page 151.

---

### Restarting Your Program and Checking for Position

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 “XRST Call” on page 184.

---

## Maintaining Database Integrity (IMS Batch, BMP, and IMS Online Regions)

IMS uses the following 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” on page 254.

Table 47 summarizes the similarities and the differences between the ROLB, ROLL, and ROLS calls.

Table 47. 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 <sup>1</sup>	X <sup>1</sup>	X <sup>1</sup>
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 <sup>2</sup>		
U3303 abnormal termination. Returns the processed input messages to the message queue.			X <sup>3</sup>
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.

**Using 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 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).

## Using ROLB

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.

***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 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; if it 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.

**In 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.

**Related Reading:** For more information on using the CMPAT keyword, see *IMS Version 9: Utilities Reference: System*. For information on coding the ROLB call, see "ROLB Call" on page 173.

## Using ROLS

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 repositioned 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. See *IMS Version 9: Administration Guide: Transaction Manager* for more information on LU 6.2.

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.

## 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. 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” on page 250.

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 following the SETS call with the same token. Figure 53 shows how the SETS and ROLS calls work together.

In addition, to assist the application program in managing other variables that it may wish to reestablish 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 is issued with the same token.

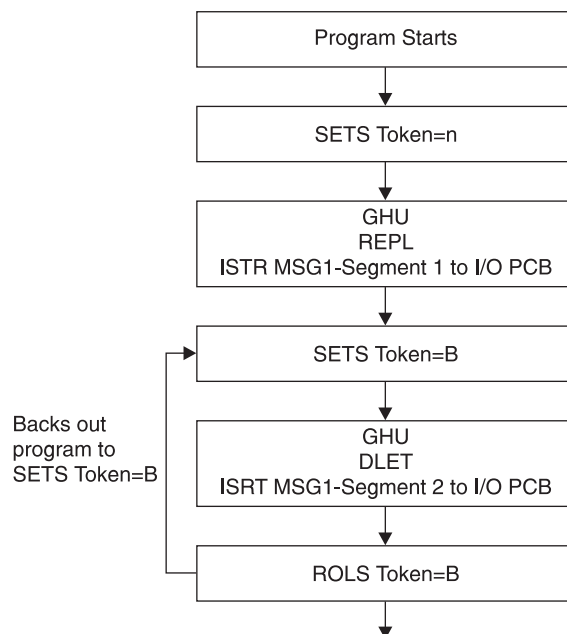


Figure 53. SETS and ROLS Calls Working Together

### Using 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 “SETS/SETU Call” on page 176.

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, see *IMS Version 9: Messages and Codes, Volume 1*. For explanations of those status codes and the response required, see *IMS Version 9: Messages and Codes, Volume 1*.

## Using 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, see *IMS Version 9: Messages and Codes, Volume 1*. For explanations of those status codes and the response required, see *IMS Version 9: Messages and Codes, Volume 1*.

---

## 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.

## 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 Q Command Code" on page 34.



## Part 2. IMS Adapter for REXX

<b>Chapter 13. IMS Adapter for REXX</b>	261
Addressing Other Environments	262
REXX Transaction Programs	262
IMS Adapter for REXX Overview Diagram	264
IVPREXX Sample Application	265
REXXTDLI Commands	266
Addressable Environments	267
REXXTDLI Calls	267
Return Codes	267
Parameter Handling	268
Example DL/I Calls	269
REXXIMS Extended Commands	270
DLIINFO	271
IMSRXTRC	272
MAPDEF	272
MAPGET	274
MAPPUT	275
SET	276
SRRBACK and SRRCMIT	277
STORAGE	278
WTO, WTP, and WTL	279
WTOR	279
IMSQUERY Extended Functions	280
<b>Chapter 14. Sample Execs Using REXXTDLI</b>	283
SAY Exec: For Expression Evaluation	283
PCBINFO Exec: Display PCBs Available in Current PSB	284
PART Execs: Database Access Example	286
PARTNUM Exec: Show Set of Parts Near a Specified Number	287
PARTNAME Exec: Show a Set of Parts with a Similar Name	287
DFSSAM01 Exec: Load the Parts Database	288
DOCMD: IMS Commands Front End	289
IVPREXX: MPP/IFP Front End for General Exec Execution	293

These topics help you use the IMS Adapter for REXX and describe addressable environments, REXX transaction programs, REXXTDLI commands and calls, and REXXIMS extended commands. The topics also provide sample EXECs using REXXTDLI.





---

## Chapter 13. IMS Adapter for REXX

The IMS adapter for REXX (REXXTDLI) provides an environment in which IMS users can interactively develop REXX EXECs under TSO/E (time-sharing option extensions) and execute them in IMS MPPs, BMPs, IFPs, or Batch regions.

This product does not compete with DFSDDLTO but is used as an adjunct. The IMS adapter for REXX provides an application programming environment for prototyping or writing low-volume transaction programs.

The REXX environment executing under IMS has the same abilities and restrictions as those documented in the *IBM TSO Extensions for MVS/REXX Reference*. These few restrictions pertain to the absence of the TSO, ISPEXEC, and ISREDIT environments, and to the absence of TSO-specific functions such as LISTDS. You can add your own external functions to the environment as documented in the *IBM TSO Extensions for MVS/REXX Reference*.

IMS calls the REXX EXEC using IRXJCL. When this method is used, Return Code 20 (RC20) is a restricted return code. Return Code 20 is returned to the caller of IRXJCL when processing was not successful, and the EXEC was not processed.

A REXX EXEC runs as an IMS application and has characteristics similar to other IMS-supported programming languages, such as COBOL. Programming language usage (REXX and other supported languages) can be mixed in MPP regions. For example, a COBOL transaction can be executed after a REXX transaction is completed, or vice versa.

REXX flexibility is provided by the following:

- REXX is an easy-to-use interpretive language.
- REXX does not require a special PSB generation to add an EXEC and run it because EXECs can run under a standard PSB (IVPREXX or one that is established by the user).
- The REXX interface supports DL/I calls and provides the following functions:
  - Call tracing of DL/I calls, status, and parameters
  - Inquiry of last DL/I call
  - Extensive data mapping
  - PCB specification by name or offset
  - Obtaining and releasing storage
  - Messaging through WTO, WTP, WTL, and WTOR

The following system environment conditions are necessary to run REXX EXECs:

- DFSREXX0 and DFSREXX1 must be in a load library accessible to your IMS dependent or batch region; for example, STEPLIB.
- DFSREXX0 is stand-alone and must have the RENT option specified.
- DFSREXX1 must be link-edited with DFSLI000 and DFSCPIR0 (for SRRCMIT and SRRBACK) and optionally, DFSREXXU. The options must be REUS, not RENT.
- IVPREXX (copy of DFSREXX0 program) must be installed as an IMS transaction program. IVP (Installation Verification Program) installs the program. For more information, see “REXX Transaction Programs” on page 262.
- The PSB must be defined as assembler language or COBOL.

- SYSEXEC DD points to a list of data sets containing the REXX EXECs that will be run in IMS. You must put this DD in your IMS dependent or batch region JCL.
- SYSTSPRT DD is used for REXX output, for example tracing, errors, and SAY instructions. SYSTSPRT DD is usually allocated as SYSOUT=A or another class, depending on installation, and must be put in the IMS dependent or batch region JCL.
- SYSTSIN DD is used for REXX input because no console exists in an IMS dependent region, as under TSO. The REXX PULL statement is the most common use of SYSTSIN.

#### **In this Chapter:**

- “Addressing Other Environments”
- “REXX Transaction Programs”
- “REXXTDLI Commands” on page 266
- “REXXTDLI Calls” on page 267
- “REXXIMS Extended Commands” on page 270

**Related Reading:** For more information on SYSTSPRT and SYSTSIN, see *IBM TSO Extensions for MVS/REXX Reference*.

---

## Addressing Other Environments

Use the REXX ADDRESS instruction to change the destination of commands. The IMS Adapter for REXX functions through two host command environments: REXXTDLI and REXXIMS. These environments are discussed in “Addressable Environments” on page 267. Other host command environments can be accessed with an IMS EXEC as well.

The z/OS environment is provided by TSO in both TSO and non-TSO address spaces. It is used to run other programs such as EXECIO for file I/O. IMS does not manage the z/OS EXECIO resources. An IMS COMMIT or BACKOUT, therefore, has no effect on these resources. Because EXECIO is not an IMS-controlled resource, no integrity is maintained. If integrity is an issue for flat file I/O, use IMS GSAM, which ensures IMS-provided integrity.

If APPC/MVS is available (MVS 4.2 or higher), other environments can be used. The environments are:

<b>APPCMVS</b>	Used for MVS-specific APPC interfacing
<b>CPICOMM</b>	Used for CPI Communications
<b>LU62</b>	Used for MVS-specific APPC interfacing

**Related Reading:** For more information on addressing environments, see *IBM TSO Extensions for MVS/REXX Reference*.

---

## REXX Transaction Programs

A REXX transaction program can use any PSB definition. The definition set up by the IVP for testing is named IVPREXX. A section of the IMS stage 1 definition is shown in the following example:

```

*****
*   IVP APPLICATIONS DEFINITION FOR DB/DC, DCCTL   *
*****
          APPLCTN GPSB=IVPREXX,PGMTYPE=TP,LANG=ASSEM  REXXTDLI SAMPLE
          TRANSACT CODE=IVPREXX,MODE=SNGL,           X
          MSGTYPE=(SNGLSEG,NONRESPONSE,1)

```

This example uses a GPSB, but you could use any PSB that you have defined. The GPSB provides a generic PSB that has an IOPCB and a modifiable alternate PCB. It does not have any database PCBs. The language type of ASSEM is specified because no specific language type exists for a REXX application.

**Recommendation:** For a REXX application, specify either Assembler language or COBOL.

IMS schedules transactions using a load module name that is the same as the PSB name being used for MPP regions or the PGM name for other region types. You must use this load module even though your application program consists of the REXX EXEC. The IMS adapter for REXX provides a load module for you to use. This module is called DFSREXX0. You can use it in one of the following ways:

- Copy to a steplib data set with the same name as the application PSB name. Use either a standard utility intended for copying load modules (such as IEBCOPY or SAS), or the Linkage Editor.
- Use the Linkage Editor to define an alias for DFSREXX0 that is the same as the application PGM name.

**Example:** Shown below is a section from the PGM setup job. It uses the linkage editor to perform the copy function to the name IVPREXX. The example uses the IVP.

```

/* REXXTDLI SAMPLE - GENERIC APPLICATION DRIVER
/*
//LINK      EXEC PGM=IEWL,
//          PARM='XREF,LIST,LET,SIZE=(192K,64K)'
//SYSPRINT  DD SYSOUT=*
//SDFSRESL  DD DISP=SHR,DSN=IMS.SDFSRESL
//SYSLMOD   DD DISP=SHR,DSN=IMS1.PGMLIB
//SYSUT1    DD UNIT=(SYSALLDA,SEP=(SYSLMOD,SYSLIN)),
//          DISP=(,DELETE,DELETE),SPACE=(CYL,(1,1))
//SYSLIN    DD *
           INCLUDE  SDFSRESL(DFSREXX0)
           ENTRY   DFSREXX0
           NAME    IVPREXX(R)
/*

```

When IMS schedules an application transaction, the load module is loaded and given control. The load module establishes the REXX EXEC name as the PGM name with an argument of the Transaction Code (if applicable). The module calls a user exit routine (DFSREXXU) if it is available. The user exit routine selects the REXX EXEC (or a different EXEC to run) and can change the EXEC arguments, or do any other desired processing.

**Related Reading:** For more information on the IMS adapter for REXX exit routine, see *IMS Version 9: Customization Guide*.

Upon return from the user exit routine, the action requested by the routine is performed. This action normally involves calling the REXX EXEC. The EXEC load occurs using the SYSEXEC DD allocation. This allocation must point to one or

more partitioned data sets containing the IMS REXX application programs that will be run as well as any functions written in REXX that are used by the programs.

Standard REXX output, such as SAY statements and tracing, is sent to SYSTSPRT. This DD is required and can be set to SYSOUT=A.

When the stack is empty, the REXX PULL statement reads from the SYSTSIN DD. In this way, you can conveniently provide batch input data to a BMP or batch region. SYSTSIN is optional; however, you will receive an error message if you issue a PULL from an empty stack and SYSTSIN is not allocated. Figure 54 shows the JCL necessary for MPP region that runs the IVPREXX sample EXEC.

```
//IVP32M11 EXEC PROC=DFSMPR,TIME=(1440),
//      AGN=IVP,          AGN NAME
//      NBA=6,
//      OBA=5,
//      SOUT='*',          SYSOUT CLASS
//      CL1=001,          TRANSACTION CLASS 1
//      CL2=000,          TRANSACTION CLASS 2
//      CL3=000,          TRANSACTION CLASS 3
//      CL4=000,          TRANSACTION CLASS 4
//      TLIM=10,         MPR TERMINATION LIMIT
//      SOD=,            SPIN-OFF DUMP CLASS
//      IMSID=IVP1,      IMSID OF IMS CONTROL REGION
//      PREINIT=DC,      PROCLIB DFSINTXX MEMBER
//      PWF1=Y           PSEUDO=WFI
//*
//* ADDITIONAL DD STATEMENTS
//*
//DFSCTL  DD DISP=SHR,
//          DSN=IVPSYS32.PROCLIB(DFSSBPRM)
//DFSSTAT DD SYSOUT=*
//* REXX EXEC SOURCE LOCATION
//SYSEXEC DD DISP=SHR,
//          DSN=IVPIVP32.INSTALIB
//          DD DISP=SHR,
//          DSN=IVPSYS32.SDFSEXEC
//* REXX INPUT LOCATION WHEN STACK IS EMPTY
//SYSTSIN DD *
//*
//* REXX OUTPUT LOCATION
//SYSTSPRT DD SYSOUT=*
//* COBOL OUTPUT LOCATION
//SYSOUT  DD SYSOUT=*
```

Figure 54. JCL Code Used to Run the IVPREXX Sample Exec

## IMS Adapter for REXX Overview Diagram

Figure 55 on page 265 shows the IMS adapter for REXX environment at a high level. This figure shows how the environment is structured under the IMS program controller, and some of the paths of interaction between the components of the environment.

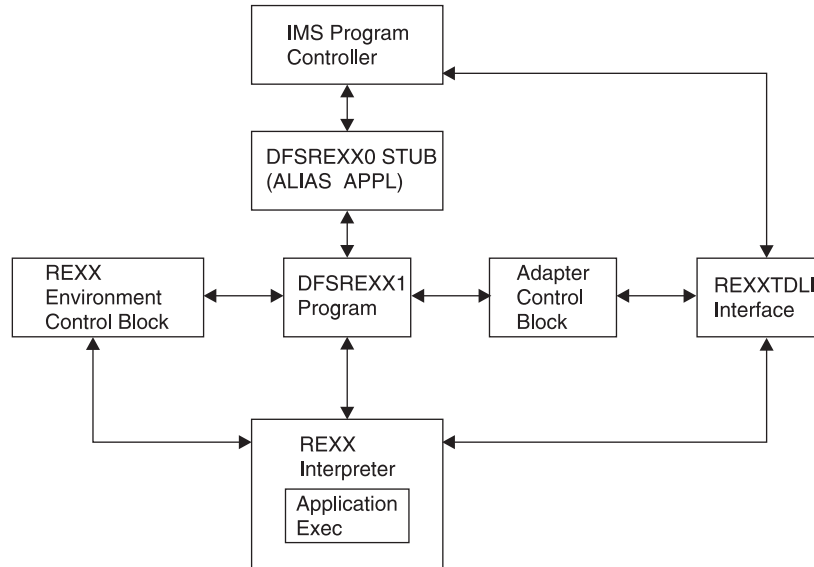


Figure 55. IMS Adapter for REXX Logical Overview Diagram

## IVPREXX Sample Application

Figure 54 on page 264 shows the JCL needed to use IVPREXX from an MPP region. This EXEC can also be run from message-driven BMPs or IFP regions.

To use the IVPREXX driver sample program in a message-driven BMP or IFP environment, specify IVPREXX as the program name and PSB name in the IMS region program's parameter list. Specifying IVPREXX loads the IVPREXX load module, which is a copy of the DFSREXX0 front-end program. The IVPREXX program loads and runs an EXEC named IVPREXX that uses message segments sent to the transaction as arguments to derive the EXEC to call or the function to perform.

Interactions with IVPREXX from an IMS terminal are shown in the following examples:

### IVPREXX Example 1

Entry:

```
IVPREXX execname
```

or

```
IVPREXX execname arguments
```

Response:

```
EXEC execname ended with RC= x
```

### IVPREXX Example 2

Entry:

```
IVPREXX LEAVE
```

Response:

```
Transaction IVPREXX leaving dependent region.
```

**IVPREXX Example 3**

Entry:

IVPREXX HELLOHELLO

Response:

One-to-eight character EXEC name must be specified.

**IVPREXX Example 4**

Entry:

IVPREXX

or

IVPREXX ?

Response:

TRANCODE EXECNAME <Arguments>	Run specified EXEC
TRANCODE LEAVE	Leave Dependent Region
TRANCODE TRACE level	0=None,1=Some,2=More,3=Full
TRANCODE ROLL	Issue ROLL call

When an EXEC name is supplied, all of the segments it inserts to the I/O PCB are returned before the completion message is returned.

REXX return codes (RC) in the range of 20000 to 20999 are usually syntax or other REXX errors, and you should check the z/OS system console or region output for more details.

**Related Reading:** For more information on REXX errors and messages, see *IBM TSO Extensions for MVS/REXX Reference*.

**Stopping an Infinite Loop:** To stop an EXEC that is in an infinite loop, you can enter either of the following IMS commands from the master terminal or system console:

```
/STO REGION p1 ABDUMP p2
/STO REGION p1 CANCEL
```

In these examples, *p1* is the region number and *p2* is the TRANCODE that the EXEC is running under. Use the /DISPLAY ACTIVE command to find the region number. This technique is not specific to REXX EXECs and can be used on any transaction that is caught in an infinite loop.

**Related Reading:** For more information about these commands and others to help in this situation, see *IMS Version 9: Command Reference*.

---

**REXXTDLI Commands**

The following section contains REXX commands and describes how they apply to DL/I calls. The terms *command* and *call* can be used interchangeably when explaining the REXXTDLI environment. However, the term *command* is used exclusively when explaining the REXXIMS environment. For consistency, *call* is used when explaining DL/I, and *command* is used when explaining REXX.

## Addressable Environments

To issue commands in the IMS adapter for REXX environment, you must first address the correct environment. Two addressable environments are provided with the IMS adapter for REXX. The environments are as follows:

**REXXTDLI** Used for standard DL/I calls, for example GU and ISRT. The REXXTDLI interface environment is used for all standard DL/I calls and cannot be used with REXX-specific commands. All commands issued to this environment are considered to be standard DL/I calls and are processed appropriately. A GU call for this environment could look like this:

```
Address REXXTDLI "GU MYPCB DataSeg"
```

**REXXIMS** Used to access REXX-specific commands (for example, WTO and MAPDEF) in the IMS adapter for REXX environment. The REXXIMS interface environment is used for both DL/I calls and REXX-specific commands. When a command is issued to this environment, IMS checks to see if it is REXX-specific. If the command is not REXX-specific, IMS checks to see if it is a standard DL/I call. The command is processed appropriately.

The REXX-specific commands, also called extended commands, are REXX extensions added by the IMS adapter for the REXX interface. A WTO call for this environment could look like this:

```
Address REXXIMS "WTO Message"
```

On entry to the scheduled EXEC, the default environment is z/OS. Consequently, you must either use ADDRESS REXXTDLI or ADDRESS REXXIMS to issue the IMS adapter for REXX calls.

**Related Reading:** For general information on addressing environments, see *IBM TSO Extensions for MVS/REXX Reference*.

---

## REXXTDLI Calls

```
►► dlicall [parm1] [parm2] [...] ►►
```

The format of a DL/I call varies depending on call type. The parameter formats for supported DL/I calls are shown in previous sections of this information. The parameters for the calls are case-independent, separated by one or more blanks, and are generally REXX variables. See "Parameter Handling" on page 268 for detailed descriptions.

## Return Codes

If you use the AIBTDLI interface, the REXX RC variable is set to the return code from the AIB on the DL/I call.

If you do not use the AIBTDLI interface, a simulated return code is returned. This simulated return code is set to zero if the PCB status code was GA, GK, or bb. If the status code had any other value, the simulated return code is X'900' or decimal 2304.

## Parameter Handling

The IMS adapter for REXX performs some parameter setup for application programs in a REXX environment. This setup occurs when the application program uses variables or maps as the parameters. When the application uses storage tokens, REXX does not perform this setup. The application program must provide the token and parse the results just as a non-REXX application would. For a list of parameter types and definitions, see Table 48.

The REXXTDLI interface performs the following setup:

- The I/O area retrieval for the I/O PCB is parsed. The LL field is removed, and the ZZ field is removed and made available by means of the REXXIMS('ZZ') function call. The rest of the data is placed in the specified variable or map. Use the REXX LENGTH() function to find the length of the returned data.
- The I/O area building for the I/O PCB or alternate PCB is done as follows:
  - The appropriate LL field.
  - The ZZ field from a preceding SET ZZ command or X'0000' if the command was not used.
  - The data specified in the passed variable or map.
- The I/O area processing for the SPA is similar to the first two items, except that the ZZ field is 4 bytes long.
- The feedback area on the CHNG and SET0 calls is parsed. The LLZZLL fields are removed, and the remaining data is returned with the appropriate length.
- The parameters that have the LLZZ as part of their format receive special treatment. These parameters occur on the AUTH, CHNG, INIT, ROLS, SET0, and SETS calls. The LLZZ fields are removed when IMS returns data to you and added (ZZ is always X'0000') when IMS retrieves data from you. In effect, your application ignores the LLZZ field and works only with the data following it.
- The numeric parameters on XRST and symbolic CHKP are converted between decimal and a 32-bit number (fullword) as required.

Table 48. IMS Adapter for REXX Parameter Types and Definitions

Type <sup>1</sup>	Parameter Definition
PCB	<p>PCB Identifier specified as a variable containing one of the following:</p> <ul style="list-style-type: none"> <li>• PCB name as defined in the PSB generation on the PCBNAME= parameter. See <i>IMS Version 9: Utilities Reference: System</i> for more information on defining PCB names. The name can be from 1 to 8 characters long and does not have to be padded with blanks. If this name is given, the AIBTDLI interface is used, and the return codes and reason codes are acquired from that interface.</li> <li>• An AIB block formatted to DFSAIB specifications. This variable is returned with an updated AIB.</li> <li>• A # followed by PCB offset number (#1=first PCB). Example settings are:                             <ul style="list-style-type: none"> <li>– IOPCB=: "#1"</li> <li>– ALTPCB=: "#2"</li> <li>– DBPCB=: "#3"</li> </ul> </li> </ul> <p>The IOAREA length returned by a database DL/I call defaults to 4096 if this notation is used. The correct length is available only when the AIBTDLI interface is used.</p>



Table 48. IMS Adapter for REXX Parameter Types and Definitions (continued)

Type <sup>1</sup>	Parameter Definition
In	Input variable. It can be specified as a constant, variable, *mapname <sup>2</sup> , or !token <sup>3</sup> .
SSA	Input variable with an SSA (segment search argument). It can be specified as a constant, variable, *mapname <sup>2</sup> , or !token <sup>3</sup> .
Out	Output variable to store a result after a successful command. It can be specified as a variable, *mapname <sup>2</sup> , or !token <sup>3</sup> .
In/Out	Variable that contains input on entry and contains a result after a successful command. It can be specified as a variable, *mapname <sup>2</sup> , or !token <sup>3</sup> .
Const	Input constant. This command argument must be the actual value, not a variable containing the value.

**Note:**

- The parameter types listed in Table 48 on page 268 correspond to the types shown (earlier in this information) under the specific DL/I calls, as well as to those shown in Table 49 on page 270..  
All parameters specified on DL/I calls are case independent except for the values associated with the STEM portion of the compound variable (REXX terminology for an array-like structure). A period (.) can be used in place of any parameter and is read as a NULL (zero length string) and written as a void (place holder). Using a period in place of a parameter is useful when you want to skip optional parameters.
- For more information on \*mapname, see "MAPGET" on page 274 and "MAPPUT" on page 275..
- For more information on !token, see "STORAGE" on page 278.

## Example DL/I Calls

The following example shows an ISRT call issued against the I/O PCB. It writes the message "Hello World."

```
IO = "IOPCB"          /* IMS Name for I/O PCB */
OutMsg="Hello World"
Address REXXTDLI "ISRT IO OutMsg"
If RC=0 Then Exit 12
```

In this example, *IO* is a variable that contains the PCB name, which is the constant "IOPCB" for the I/O PCB. If a non-zero return code (RC) is received, the EXEC ends (Exit) with a return code of 12. You can do other processing here.

The next example gets a part from the IMS sample parts database. The part number is "250239". The actual part keys have a "02" prefix and the key length defined in the DBD is 17 bytes.

The following example puts the segment into the variable called *Part\_Segment*.

```
PartNum = "250239"
DB      = "DBPCB01"
SSA     = 'PARTROOT(PARTKEY = '|Left('02'|PartNum,17)|'|)'
Address REXXTDLI "GU DB Part_Segment SSA"
```

**Notes:**

- In a real EXEC, you would probably find the value for PartNum from an argument and would have to check the return code after the call.

- The LEFT function used here is a built-in REXX function. These built-in functions are available to any IMS REXX EXEC. For more information on functions, see *IBM TSO Extensions for MVS/REXX Reference*.
- The single quote (') and double quote (") are interchangeable in REXX, as long as they are matched.

The IMS.SDFSISRC library includes the DFSSUT04 EXEC. You can use this EXEC to process any unexpected return codes or status codes. To acquire the status code from the last DL/I call issued, you must execute the IMSQUERY('STATUS') function. It returns the two character status code.

### Environment Determination

If you use an EXEC that runs in both IMS and non-IMS environments, check to see if the IMS environment is available. You can check to see if the IMS environment is available in two ways:

- Use the z/OS SUBCOM command and specify either the REXXTDLI or REXXIMS environments. The code looks like this:

```
Address z/OS 'SUBCOM REXXTDLI'
If RC=0 Then Say "IMS Environment is Available."
Else Say "Sorry, no IMS Environment here."
```

- Use the PARSE SOURCE instruction of REXX to examine the address space name (the 8th word). If it is running in an IMS environment, the token will have the value IMS. The code looks like this:

```
Parse Source . . . . . Token .
If Token='IMS' Then Say "IMS Environment is Available."
Else Say "Sorry, no IMS Environment here."
```

---

## REXXIMS Extended Commands

The IMS adapter for REXX gives access to the standard DL/I calls and it supplies a set of extended commands for the REXX environment. These commands are listed in Table 49 and are available when you ADDRESS REXXIMS. DL/I calls are also available when you address the REXXIMS environment.

Table 49 shows the extended commands. The following pages contain detailed descriptions of each command.

Table 49. REXXIMS Extended Commands

Command	Parameter Types <sup>1</sup>
DLIINFO	Out [PCB]
IMSRXTRC	In
MAPDEF	Const In [Const]
MAPGET	Const In
MAPPUT	Const Out
SET	Const In
SRRBACK	Out
SRRCMIT	Out
STORAGE	Const Const [In [Const] ]
WTO	In
WTP	In
WTL	In

Table 49. REXXIMS Extended Commands (continued)

Command	Parameter Types <sup>1</sup>
WTOR	In Out

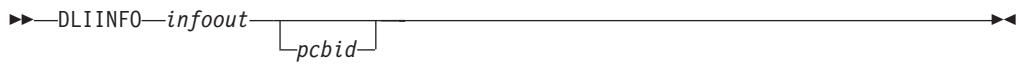
**Note:**

- The parameter types listed correspond to the types shown in Table 48 on page 268. All parameters specified on DL/I calls are case-independent except for the values associated with the STEM portion of the compound variable (REXX terminology for an array-like structure). A period (.) can be used in place of any parameter and has the effect of a NULL (zero length string) if read and a void (place holder) if written. Use a period in place of a parameter to skip optional parameters.

## DLIINFO

The DLIINFO call requests information from the last DL/I call or on a specific PCB.

### Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
DLIINFO	X	X	X	X	X

### Usage

The *infoout* variable name is a REXX variable that is assigned the DL/I information. The *pcbld* variable name, when specified as described in “Parameter Handling” on page 268, returns the addresses associated with the specified PCB and its last status code.

The format of the returned information is as follows:

### Word Description

- 1 Last DL/I call ( '.' if N/A)
- 2 Last DL/I PCB name (name or #number, '.' if N/A)
- 3 Last DL/I AIB address in hexadecimal (00000000 if N/A)
- 4 Last DL/I PCB address in hexadecimal (00000000 if N/A)
- 5 Last DL/I return code (0 if N/A)
- 6 Last DL/I reason code (0 if N/A)
- 7 Last DL/I call status ( '.' if blank or N/A)

### Example

```
Address REXXIMS 'DLIINFO MyInfo'          /* Get Info          */
Parse Var MyInfo DLI_Cmd DLI_PCB DLI_AIB_Addr DLI_PCB_Addr,
          DLI_RC DLI_Reason DLI_Status .
```

Always code a period after the status code (seventh word returned) when parsing to allow for transparent additions in the future if needed. Words 3, 4, and 7 can be used when a *pcbld* is specified on the DLIINFO call.

## IMSRXTRC

The IMSRXTRC command is used primarily for debugging. It controls the tracing action taken (that is, how much trace output through SYSTSPRT is sent to the user) while running a REXX program.

### Format

►►—IMSRXTRC—*level*—————►►

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
IMSRXTRC	X	X	X	X	X

### Usage

The *level* variable name can be a REXX variable or a digit, and valid values are from 0 to 9. The initial value at EXEC start-up is 1 unless it is overridden by the user Exit. Traced output is sent to the DDNAME SYSTSPRT. See *IMS Version 9: Customization Guide* for more information on the IMS adapter for REXX exit routine.

The IMSRXTRC command can be used in conjunction with or as a replacement for normal REXX tracing (TRACE).

### Level Description

- 0** Trace errors only.
- 1** The previous level and trace DL/I calls, their return codes, and environment status (useful for flow analysis).
- 2** All the previous levels and variable sets.
- 3** All the previous levels and variable fetches (useful when diagnosing problems).
- 4-7** All previous levels.
- 8** All previous levels and parameter list to/from standard IMS language interface. See message DFS3179 in *IMS Version 9: Messages and Codes, Volume 1*.
- 9** All previous levels.

### Example

```
Address REXXIMS 'IMSRXTRC 3'
```

IMSRXTRC is independent of the REXX TRACE instruction.

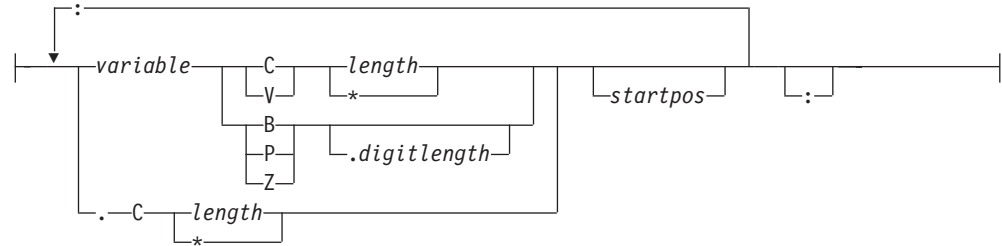
## MAPDEF

The MAPDEF command makes a request to define a data mapping.

### Format

►►—MAPDEF—*mapname*—| A | REPLACE—————►►

**A:**



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
MAPDEF	X	X	X	X	X

**Usage**

Data mapping is an enhancement added to the REXXIMS interface. Because REXX does not offer variable structures, parsing the fields from your database segments or MFS output maps can be time consuming, especially when data conversion is necessary. The MAPDEF, MAPGET, and MAPPUT commands allow simple extraction of most formatted data.

- *mapname* is a 1- to 16-character case-independent name.
- definition (**A**) is a variable containing the map definition.
- REPLACE, if specified, indicates that a replacement of an existing map name is allowed. If not specified and the map name is already defined, an error occurs and message DFS3171E is sent to the SYSTPRT.

The map *definition* has a format similar to data declarations in other languages, with simplifications for REXX. In this definition, you must declare all variables that you want to be parsed with their appropriate data types. The format is shown in **A** in the syntax diagram.

**Variable name:** The variable name *variable* is a REXX variable used to contain the parsed information. Variable names are case-independent. If you use a STEM (REXX terminology for an array-like structure) variable, it is resolved at the time of use (at the explicit or implicit MAPGET or MAPPUT call time), and this can be very powerful. If you use an index type variable as the STEM portion of a compound variable, you can load many records into an array simply by changing the index variable. Map names or tokens cannot be substituted for variable names inside a map definition.

**Repositioning the internal cursor:** A period (.) can be used as a variable place holder for repositioning the internal cursor position. In this case, the data type must be C, and the length can be negative, positive, or zero. Use positive values to skip over fields of no interest. Use negative lengths to redefine fields in the middle of a map without using absolute positioning.

The data type values are:

- C** Character
- V** Variable
- B** Binary (numeric)
- Z** Zoned Decimal (numeric)

**P** Packed Decimal (numeric)

All numeric data types can have a period and a number next to them. The number indicates the number of digits to the right of a decimal point when converting the number.

**Length value:** The *length* value can be a number or an asterisk (\*), which indicates that the rest of the buffer will be used. You can only specify the *length* value for data types C and V. Data type V maps a 2-byte length field preceding the data string, such that a when the declared length is 2, it takes 4 bytes.

Valid lengths for data types are:

<b>C</b>	1 to 32767 bytes or *
<b>V</b>	1 to 32765 bytes or *
<b>B</b>	1 to 4 bytes
<b>Z</b>	1 to 12 bytes
<b>P</b>	1 to 6 bytes

If a value other than asterisk (\*) is given, the cursor position is moved by that value.

The *startpos* value resets the parsing position to a fixed location. If *startpos* is omitted, the column to the right of the previous map variable definition (cursor position) is used. If it is the first variable definition, column 1 is used.

**Note:** A length of asterisk (\*) does not move the cursor position, so a variable declared after one with a length of asterisk (\*) without specifying a start column overlays the same definition.

**Example**

This example defines a map named DBMAP, which is used implicitly on a GU call by placing an asterisk (\*) in front of the map name.

```
DBMapDef = 'RECORD      C   * :', /* Pick up entire record      */
          'NAME        C  10 :', /* Cols 1-10 hold the name */
          'PRICE       Z.2 6 :', /* Cols 11-16 hold the price */
          'CODE        C   2 :', /* Cols 11-16 hold the code */
          '            C  25 :', /* Skip 25 columns        */
          'CATEGORY    B   1' /* Col 42 holds category   */
Address REXXIMS 'MAPDEF DBMAP DBMapDef'

:
:
Address REXXTDLI 'GU DBPCB *DBMAP' /* Read and decode a segment */
If RC=0 Then Signal BadCall      /* Check for failure          */
Say CODE                          /* Can now access any Map Variable*/
```

The entire segment retrieved on the GU call is placed in RECORD. The first 10 characters are placed in NAME, and the next 6 are converted from zoned decimal to EBCDIC with two digits to the right of the decimal place and placed in PRICE. The next 2 characters are placed in CODE, the next 25 are skipped, and the next character is converted from binary to EBCDIC and placed in CATEGORY. The 25 characters that are skipped are present in the RECORD variable.

**MAPGET**

The MAPGET command is a request to parse or convert a buffer into a specified data mapping previously defined with the MAPDEF command.

## Format

►►—MAPGET—*mapname*—*buffer*—◄◄

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
MAPGET	X	X	X	X	X

## Usage

The *mapname* variable name specifies the data mapping to use. It is a 1- to 16-character case-independent name. The *buffer* variable name is the REXX variable containing the data to parse.

Map names can also be specified in the REXXTDLI calls in place of variable names to be set or written. This step is called an implicit MAPGET. Thus, the explicit (or variable dependent) MAPGET call can be avoided. To indicate that a Map name is being passed in place of a variable in the DL/I call, precede the name with an asterisk (\*), for example, 'GU IOPCB \*INMAP'.

## Examples

This example uses explicit support.

```
Address REXXTDLI 'GU DBPCB SegVar'
If RC=0 Then Signal BadCall          /* Check for failure          */
Address REXXIMS 'MAPGET DBMAP SegVar' /* Decode Segment            */
Say VAR_CODE                          /* Can now access any Map Variable */
```

This example uses implicit support.

```
Address REXXTDLI 'GU DBPCB *DBMAP' /* Read and decode segment if read*/
If RC=0 Then Signal BadCall          /* Check for failure          */
Say VAR_CODE                          /* Can now access any Map Variable*/
```

If an error occurs during a MAPGET, message DFS3172I is issued. An error could occur when a Map is defined that is larger than the input segment to be decoded or during a data conversion error from packed or zoned decimal format. The program continues, and an explicit MAPGET receives a return code 4. However, an implicit MAPGET (on a REXXTDLI call, for example) does not have its return code affected. Either way, the failing variable's value is dropped by REXX.

## MAPPUT

This MAPPUT command makes a request to pack or concatenate variables from a specified Data Mapping, defined by the MAPDEF command, into a single variable.

## Format

►►—MAPPUT—*mapname*—*buffer*—◄◄

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
MAPPUT	X	X	X	X	X

## Usage

The *mapname* variable name specifies the data mapping to use, a 1- to 16-character case-independent name. The *buffer* variable name is the REXX variable that will contain the resulting value.

Map names can also be specified in the REXXTDLI call in place of variable names to be fetched or read. This step is called an implicit MAPPUT and lets you avoid the explicit MAPPUT call. To indicate that a Map name is being passed in the DL/I call, precede the name with an asterisk (\*), for example, 'ISRT IOPCB \*OUTMAP'.

**Note:** If the data mapping is only partial and some fields in the record are not mapped to REXX variables, then the first field in the mapping should be a character type of length asterisk (\*), as shown in the “Example” on page 274. This step is the only way to ensure that non-mapped (skipped) fields are not lost between the MAPGET and MAPPUT calls, whether they be explicit or implicit.

### Examples

This example uses explicit support.

```
Address REXXTDLI
'GHU DBPCB SegVar SSA1'           /* Read segment          */
If RC=0 Then Signal BadCall        /* Check for failure     */
Address REXXIMS 'MAPGET DBMAP SegVar' /* Decode Segment       */
DBM_Total = DBM_Total + Deposit_Amount /* Adjust Mapped Variable */
Address REXXIMS 'MAPPUT DBMAP SegVar' /* Encode Segment       */
'REPL DBPCB SegVar'              /* Update Database       */
If RC=0 Then Signal BadCall        /* Check for failure     */
```

This example uses implicit support.

```
Address REXXTDLI
'GHU DBPCB *DBMAP SSA1'           /* Read and decode segment if read */
If RC=0 Then Signal BadCall        /* Check for failure     */
DBM_Total = DBM_Total + Deposit_Amount /* Adjust Mapped Variable */
'REPL DBPCB *DBMAP'              /* Update Database       */
If RC=0 Then Signal BadCall        /* Check for failure     */
```

If an error occurs during a MAPPUT, such as a Map field defined larger than the variable's contents, then the field is truncated. If the variable's contents are shorter than the field, the variable is padded:

**Character (C)** Padded on right with blanks  
**Character (V)** Padded on right with zeros  
**Numeric (B,Z,P)** Padded on the left with zeros

If a MAP variable does not exist when a MAPPUT is processed, the variable and its position are skipped. All undefined and skipped fields default to binary zeros. A null parameter is parsed normally. Conversion of non-numeric or null fields to numeric field results in a value of 0 being used and no error.

## SET

The SET command resets AIB subfunction values and ZZ values before you issue a DL/I call.

### Format

```
➤➤ SET SUBFUNC variable
      ZZ variable
```

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
SET	X	X	X	X	X



**Usage**

The SET SUBFUNC command sets the AIB subfunction used on the next DL/I call. This value is used only if the next REXXTDLI call passes a PCB name. If the call does pass a PCB name, the IMS adapter for REXX places the subfunction name (1 to 8 characters or blank) in the AIB before the call is issued. This value initially defaults to blanks and is reset to blanks on completion of any REXXTDLI DL/I call. For more information on subfunctions, see the appropriate sections in this information.

The SET ZZ command is used to set the ZZ value used on a subsequent DL/I call. This command is most commonly used in IMS conversational transactions and terminal dependent applications to set the ZZ field to something other than the default of binary zeros. Use the SET command before an ISRT call that requires other than the default ZZ value. For more explanation on ZZ processing, see "Parameter Handling" on page 268.

**Examples**

This example shows the SET SUBFUNC command used with the INQY call to get environment information.

```
IO="IOPCB"
Func = "ENVIRON"           /* Sub-Function Value */
Address REXXIMS "SET SUBFUNC Func" /* Set the value */
Address REXXTDLI "INQY IO EnviData" /* Make the DL/I Call */
IMS_Identifier = Substr(EnviData,1,8) /* Get IMS System Name*/
```

This example shows the SET ZZ command used with a conversational transaction for SPA processing.

```
Address REXXTDLI 'GU IOPCB SPA' /* Get first Segment */
Hold_ZZ = IMSQUERY('ZZ') /* Get ZZ Field (4 bytes) */
:
Address REXXIMS 'SET ZZ Hold_ZZ' /* Set ZZ for SPA ISRT */
Address REXXTDLI 'ISRT IOPCB SPA' /* ISRT the SPA */
```

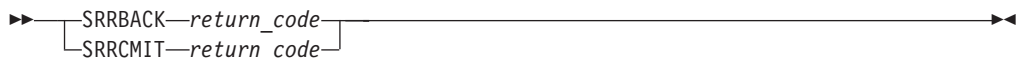
This example shows the SET ZZ command used for setting 3270 Device Characteristics Flags.

```
Bell_ZZ = '0040'X /* ZZ to Ring Bell on Term */
Address REXXIMS 'SET ZZ Bell_ZZ' /* Set ZZ for SPA ISRT */
Address REXXTDLI 'ISRT IOPCB Msg' /* ISRT the Message */
```

**SRRBACK and SRRCMIT**

The Common Programming Interface Resource Recovery (CPI-RR) commands allow an interface to use the SAA<sup>®</sup> resource recovery interface facilities for back-out and commit processing.

**Format**



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
SRRBACK, SRRCMIT	X		X		

## Usage

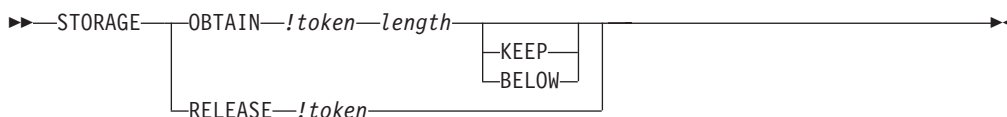
The return code from the SRR command is returned and placed in the *return\_code* variable name as well as the REXX variable RC.

For more information on SRRBACK and SRRCMIT, see *IMS Version 9: Administration Guide: Transaction Manager and System Application Architecture Common Programming Interface: Resource Recovery Reference*.

## STORAGE

The STORAGE command allows the acquisition of system storage that can be used in place of variables for parameters to REXXTDLI and REXXIMS calls.

### Format



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
STORAGE	X	X	X	X	X

## Usage

Although REXX allows variables to start with characters (!) and (#), these characters have special meanings on some commands. When using the REXXTDLI interface, you must not use these characters as the starting characters of variables.

The *!token* variable name identifies the storage, and it consists of an exclamation mark followed by a 1- to 16-character case-independent token name. The *length* variable name is a number or variable containing size in decimal to OBTAIN in the range 4 to 16777216 bytes (16 MB). The storage class has two possible override values, BELOW and KEEP, of which only one can be specified for any particular token. The BELOW function acquires the private storage below the 16 MB line. The KEEP function marks the token to be kept after this EXEC is terminated. The default action gets the storage in any location and frees the token when the EXEC is terminated.

Use the STORAGE command to get storage to use on DL/I calls when the I/O area must remain in a fixed location (for example, Spool API) or when it is not desirable to have the LLZZ processing. For more information on LLZZ processing, see "Parameter Handling" on page 268. Once a token is allocated, you can use it in REXXTDLI DL/I calls or on the STORAGE RELEASE command.

Note the following when using STORAGE:

- When used on DL/I calls, none of the setup for LLZZ fields takes place. You must fill the token in and parse the results from it just as required by a non-REXX application.
- You cannot specify both KEEP and BELOW on a single STORAGE command.
- The RELEASE function is only necessary for tokens marked KEEP. All tokens not marked KEEP and not explicitly released by the time the EXEC ends are released automatically by the IMS adapter for REXX.
- When you use OBTAIN, the entire storage block is initialized to 0.

- The starting address of the storage received is always on the boundary of a double word.
- You cannot re-obtain a token until RELEASE is used or the EXEC that obtained it, non-KEEP, terminates. If you try, a return code of -9 is given and the error message DFS3169 is issued.
- When KEEP is specified for the storage token, it can be accessed again when this EXEC or another EXEC knowing the token's name is started in the same IMS region.
- Tokens marked KEEP are not retained when an ABEND occurs or some other incident occurs that causes region storage to be cleared. It is simple to check if the block exists on entry with the IMSQUERY(!token) function. For more information, see "IMSQUERY Extended Functions" on page 280.

**Example**

This example shows how to use the STORAGE command with Spool API.

```

/* Get 4K Buffer below the line for Spool API Usage */
Address REXXIMS 'STORAGE OBTAIN !MYTOKEN 4096 BELOW'
/* Get Address and length (if curious) */
Parse Value IMSQUERY(!MYTOKEN) With My_Token_Addr My_Token_Len.
Address REXXIMS 'SETO ALTPCB !MYTOKEN SETOPARMS SETOFB'

:
:
Address REXXIMS 'STORAGE RELEASE !MYTOKEN'
    
```

**WTO, WTP, and WTL**

The WTO command is used to write a message to the operator. The WTP command is used to write a message to the program (WTO ROUTCDE=11). The WTL command is used to write a message to the console log.

**Format**



Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
WTO, WTP, WTL	X	X	X	X	X

**Usage**

The *message* variable name is a REXX variable containing the text that is stored displayed in the appropriate place.

**Example**

This example shows how to write a simple message stored the REXX variable MSG.

```

Msg = 'Sample output message.'          /* Build Message          */
Address REXXIMS 'WTO Msg'                /* Tell Operator          */
Address REXXIMS 'WTP Msg'                /* Tell Programmer        */
Address REXXIMS 'WTL Msg'                /* Log It                 */
    
```

**WTOR**

The WTOR command requests input or response from the z/OS system operator.

### Format

▶▶—WTOR—*message*—*response*—▶▶

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
WTOR	X	X	X	X	X

### Usage

The *message* variable name is a REXX variable containing the text that will be displayed on the z/OS console. The operator's response is placed in the REXX variable signified by the *response* variable name.

**Attention:** This command hangs the IMS region in which it is running until the operator responds.

### Example

This example prompts the operator to enter ROLL or CONT on the z/OS master or alternate console. Once the WTOR is answered, the response is placed in the REXX variable name *response*, and the EXEC will process the IF statement appropriately.

```
Msg = 'Should I ROLL or Continue. Reply "ROLL" or "CONT"'
Address REXXIMS 'WTOR Msg Resp'           /* Ask Operator */
If Resp = 'ROLL' Then                     /* Tell Programmer */
    Address REXXTDLI 'ROLL'               /* Roll Out of this */
```

## IMSQUERY Extended Functions

The IMSQUERY function is available to query certain IMS information either on the environment or on the prior DL/I call.

### Format

▶▶—IMSQUERY—(—  
 FEEDBACK—  
 IMSRXTRC—  
 REASON—  
 SEGLEVEL—  
 SEGNAME—  
 STATUS—  
 TRANCODE—  
 USERID—  
 ZZ—  
 !token—  
 )—▶▶

Call Name	DB/DC	DBCTL	DCCTL	DB Batch	TM Batch
IMSQUERY	X	X	X	X	X

### Usage

The format of the function call is: IMSQUERY('Argument') where Argument is one of the following values:

Argument	Description of Data Returned
FEEDBACK	FEEDBACK area from current PCB.
IMSRXTRC	Current IMSRXTRC trace level #.

<b>REASON</b>	Reason code from last call (from AIB if used on last REXXTDLI type call).
<b>SEGLEVEL</b>	Segment level from current PCB (Last REXXTDLI call must be against a DB PCB, or null is returned).
<b>SEGNAME</b>	Segment name from current PCB (Last REXXTDLI call must be against a DB PCB, or null is returned).
<b>STATUS</b>	IMS status code from last executed REXXTDLI call (DL/I call). This argument is the two character status code from the PCB.
<b>TRANCODE</b>	Current transaction code being processed, if available.
<b>USERID</b>	Input terminal's user ID, if available. If running in a non-message-driven region, the value is dependent on the specification of the BMPUSID= keyword in the DFSDCxxx PROCLIB member: <ul style="list-style-type: none"> <li>• If BMPUSID=USERID is specified, the value from the USER= keyword on the JOB statement is used.</li> <li>• If USER= is not specified on the JOB statement, the program's PSB name is used.</li> <li>• If BMPUSID=PSBNAME is specified, or if BMPUSID= is not specified at all, the program's PSB name is used.</li> </ul>
<b>ZZ</b>	ZZ (of LLZZ) from last REXXTDLI command. This argument can be used to save the ZZ value after you issue a GU call to the I/O PCB when the transaction is conversational.
<b>!token</b>	Address (in hexadecimal) and length of specified token (in decimal), separated by a blank.

This value can be placed in a variable or resolved from an expression. In these cases, the quotation marks should be omitted as shown below:

```
Token_Name="!MY_TOKEN"
AddrInfo=IMSQUERY(Token_Name)
/* or */
AddrInfo=IMSQUERY("!MY_TOKEN")
```

Although the function argument is case-independent, no blanks are allowed within the function argument. You can use the REXX STRIP function on the argument, if necessary. IMSQUERY is the preferred syntax, however REXXIMS is supported and can be used, as well.

### Example

```
If REXXIMS('STATUS')='GB' Then Signal End_Of_DB
:
:
Hold_ZZ = IMSQUERY('ZZ') /* Get current ZZ field*/
:
:
Parse Value IMSQUERY('!MYTOKEN') With My_Token_Addr My_Token_Len .
```

**Related Reading:** For information on the IMS adapter for REXX exit routine, see *IMS Version 9: Customization Guide*.



---

## Chapter 14. Sample Execs Using REXXTDLI

This chapter shows samples of REXX execs that use REXXTDLI to access IMS services.

The example sets are designed to highlight various features of writing IMS applications in REXX. The samples in this section are simplified and might not reflect actual usage (for example, they do not use databases).

The PART exec database access example is a set of three execs that access the PART database, which is built by the IMS installation verification program (IVP). The first two execs in this example, PARTNUM and PARTNAME, are extensions of the PART transaction that runs the program DFSSAM02, which is supplied with IMS as part of IVP. The third exec is the DFSSAM01 exec supplied with IMS and is an example of the use of EXECIO within an exec.

### **In this Chapter:**

- “SAY Exec: For Expression Evaluation”
- “PCBINFO Exec: Display PCBs Available in Current PSB” on page 284
- “PART Execs: Database Access Example” on page 286
- “DOCMD: IMS Commands Front End” on page 289
- “IVPREXX: MPP/IFP Front End for General Exec Execution” on page 293

---

### **SAY Exec: For Expression Evaluation**

Figure 56 is a listing of the SAY exec. SAY evaluates an expression supplied as an argument and displays the results. The REXX command INTERPRET is used to evaluate the supplied expression and assign it to a variable. Then that variable is used in a formatted reply message.

```

/* EXEC TO DO CALCULATIONS */
Address REXXTDLI
Arg Args
If Args='' Then
  Msg='SUPPLY EXPRESSION AFTER EXEC NAME.'
Else Do
  Interpret 'X='Args          /* Evaluate Expression */
  Msg='EXPRESSION:' Args '=' X
End
'ISRT IOPCB MSG'
Exit RC

```

*Figure 56. Exec To Do Calculations*

This exec shows an example of developing applications with IMS Adapter for REXX. It also shows the advantages of REXX, such as dynamic interpretation, which is the ability to evaluate a mathematical expression at run-time.

A PDF EDIT session is shown in Figure 57 on page 284. This figure shows how you can enter a new exec to be executed under IMS.

```

EDIT ---- USER.PRIVATE.PROCLIB(SAY) - 01.03 ----- COLUMNS 001 072
COMMAND ==> SCROLL ==> PAGE
***** ***** TOP OF DATA *****
000001 /* EXEC TO DO CALCULATIONS */
000002 Address REXXTDLI
000003 Arg Args
000004 If Args='' Then
000005     Msg='SUPPLY EXPRESSION AFTER EXEC NAME.'
000006 Else Do
000007     Interpret 'X'=Args          /* Evaluate Expression */
000008     Msg='EXPRESSION:' Arg= ' ' X
000009 End
000010
000011 'ISRT IOPCB MSG'
000012 Exit RC
***** ***** BOTTOM OF DATA *****

```

Figure 57. PDF EDIT Session on the SAY Exec

To execute the SAY exec, use IVPREXX and supply an expression such as:

```
IVPREXX SAY 5*5+7
```

This expression produces the output shown in Figure 58.

```

EXPRESSION: 5*5+7 = 32
EXEC SAY ended with RC= 0

```

Figure 58. Example Output from the SAY Exec

## PCBINFO Exec: Display PCBs Available in Current PSB

The PCB exec maps the PCBs available to the exec, which are the PCBs for the executing PSB. The mapping consists of displaying the type of PCB (IO, TP, or DB), the LTERM or DBD name that is associated, and other useful information. Mapping displays this information by using the PCB function described in “DLIINFO” on page 271. Example output screens are shown in Figure 59 and Figure 60. The listing is shown in Figure 61 on page 285. PCB mappings are created by placing DFSREXX0 in an early concatenation library and renaming it to an existing application with a PSB/DBD generation.

```

IMS PCB System Information Exec: PCBINFO
System Date: 09/26/92   Time: 15:52:15

PCB # 1: Type=IO, LTERM=T3270LC Status=   UserID=       OutDesc=DFSM02
          Date=91269 Time=1552155
PCB # 2: Type=TP, LTERM=* NONE * Status=AD
PCB # 3: Type=TP, LTERM=* NONE * Status=
PCB # 4: Type=TP, LTERM=CTRL  Status=
PCB # 5: Type=TP, LTERM=T3275  Status=
EXEC PCBINFO ended with RC= 0

```

Figure 59. Example Output of PCBINFO Exec on a PSB without Database PCBs.

```

IMS PCB System Information Exec: PCBINFO
System Date: 09/26/92   Time: 15:53:34

PCB # 1: Type=IO, LTERM=T3270LC Status=   UserID=       OutDesc=DFSM02
          Date=89320 Time=1553243
PCB # 2: Type=DB, DBD =DI21PART Status=   Level=00 Opt=G
EXEC PCBINFO ended with RC= 0

```

Figure 60. Example Output of PCBINFO Exec on a PSB with a Database PCB.



```

/* REXX EXEC TO SHOW SYSTEM LEVEL INFO */
Address REXXTDLI
Arg Dest .
WTO=(Dest='WTO')
Call SayIt 'IMS PCB System Information Exec: PCBINFO'
Call SayIt 'System Date:' Date('U') ' Time:' Time()
Call Sayit ' '
/* A DFS3162 message is given when this exec is run because it does */
/* not know how many PCBs are in the list and it runs until it gets */
/* an error return code. Note this does not show PCBs that are */
/* available to the PSB by name only, that is, not in the PCB list. */
Msg='PCBINFO: Error message normal on DLIINFO.'
'WTP MSG'
Do i=1 by 1 until Result='LAST'
    Call SayPCB i
End
Exit 0

```

SayPCB: Procedure Expose WTO

```

Arg PCB
'DLIINFO DLIINFO #'PCB /* Get PCB Address */
If rc<0 Then Return 'LAST' /* Invalid PCB Number */
Parse Var DLIInfo . . AIBAddr PCBAddr .
PCBINFO=Storage(PCBAddr,255) /* Read PCB */
DCPCB=(Substr(PCBInfo,13,1)='00'x) /* Date Field, must be DC PCB */
If DCPCB then Do
    Parse Value PCBInfo with,
        LTERM 9 . 11 StatCode 13 CurrDate 17 CurrTime 21,
        InputSeq 25 OutDesc 33 UserID 41
    If LTERM='' then LTERM='* NONE *'
    CurrDate=Substr(c2x(CurrDate),3,5)
    CurrTime=Substr(c2x(CurrTime),1,7)
    If CurrDate~='000000' then Do
        Call SayIt 'PCB #'Right(PCB,2)': Type=IO, LTERM=LTERM,
            'Status=StatCode 'UserID=UserID 'OutDesc=OutDesc
        Call SayIt ' Date=CurrDate 'Time=CurrTime
    End
    Else
        Call SayIt 'PCB #'Right(PCB,2)': Type=TP, LTERM=LTERM,
            'Status=StatCode
End
Else Do
    Parse Value PCBInfo with,
        DBDName 9 SEGLev 11 StatCode 13 ProcOpt 17 . 21 Segname . 29,
        KeyLen 33 NumSens 37
    KeyLen = c2d(KeyLen)
    NumSens= c2d(NumSens)

    Call SayIt 'PCB #'Right(PCB,2)': Type=DB, DBD =DBDName,
        'Status=StatCode 'Level=SegLev 'Opt=ProcOpt
End
Return '

```

SayIt: Procedure Expose WTO

```

Parse Arg Msg
If WTO Then
    'WTO MSG'
Else
    'ISRT IOPCB MSG'
Return

```

Figure 61. PCBINFO Exec Listing

## PART Execs: Database Access Example

This set of execs accesses the PART database shipped with IMS. These execs demonstrate fixed-record database reading, SSAs, and many REXX functions. The PART database execs (PARTNUM, PARTNAME, and DFSSAM01) are described in this section.

The PARTNUM exec is used to show part numbers that begin with a number equal to or greater than the number you specify. An example output screen is shown in Figure 62.

To list part numbers beginning with the number “300” or greater, enter the command:

```
PARTNUM 300
```

All part numbers that begin with a 300 or larger numbers are listed. The listing is shown in Figure 64 on page 287.

```
IMS Parts DATABASE Transaction
System Date: 02/16/92   Time: 23:28:41

Request: Display 5 Parts with Part_Number >= 300
 1 Part=3003802         Desc=CHASSIS
 2 Part=3003806         Desc=SWITCH
 3 Part=3007228         Desc=HOUSING
 4 Part=3008027         Desc=CARD FRONT
 5 Part=3009228         Desc=CAPACITOR

EXEC PARTNUM ended with RC= 0
```

Figure 62. Example Output of PARTNUM Exec

PARTNAME is used to show part names that begin with a specific string of characters.

To list part names beginning with “TRAN”, enter the command:

```
PARTNAME TRAN
```

All part names that begin with “TRAN” are listed on the screen. The screen is shown in Figure 63. The listing is shown in Figure 65 on page 288.

```
IMS Parts DATABASE Transaction
System Date: 02/16/92   Time: 23:30:09

Request: Display 5 Parts with Part Name like TRAN
 1 Part=250239          Desc=TRANSISTOR
 2 Part=7736847P001     Desc=TRANSFORMER
 3 Part=975105-001     Desc=TRANSFORMER
 4 Part=989036-001     Desc=TRANSFORMER
End of DataBase reached before 5 records shown.

EXEC PARTNAME ended with RC= 0
```

Figure 63. Example Output of PARTNAME Exec

The DFSSAM01 exec is used to load the parts database. This exec is executed in batch, is part of the IVP, and provides an example of EXECIO usage in an exec.

**Related Reading:** For details, see *IMS Version 9: Installation Volume 1: Installation Verification*.

## PARTNUM Exec: Show Set of Parts Near a Specified Number

**Requirement:** The following REXX exec is designed to be run by the IVPREXX exec with PSB=DFSSAM02.

```

/* REXX EXEC TO SHOW A SET OF PARTS NEAR A SPECIFIED NUMBER */
/* Designed to be run by the IVPREXX exec with PSB=DFSSAM02 */
/* Syntax: IVPREXX PARTNUM string <start#> */

Address REXXTDLI
IOPCB='IOPCB' /* PCB Name */
DataBase='#2' /* PCB # */
RootSeg_Map = 'PNUM C 15 3 : DESCRIPTION C 20 27'
'MAPDEF ROOTSEG ROOTSEG_MAP'
Call SayIt 'IMS Parts DATABASE Transaction'
Call SayIt 'System Date:' Date('U') ' Time:' Time()
Call Sayit ' '

Arg PartNum Segs .
If ~DataType(Segs,'W') then Segs=5 /* default view amount */

PartNum=Left(PartNum,15) /* Pad to 15 with Blanks */
If PartNum='' then
  Call Sayit 'Request: Display first' Segs 'Parts in the DataBase'
Else
  Call Sayit 'Request: Display' Segs 'Parts with Part_Number >=' PartNum
  SSA1='PARTROOT(PARTKEY >=02'PartNum')'
  'GU DATABASE *ROOTSEG SSA1'
  Status=IMSQUERY('STATUS')
  If Status='GE' then Do /* Segment Not Found */
    Call Sayit 'No parts found with larger Part_Number'
    Exit 0
  End
  Do i=1 to Segs While Status=' '
    Call Sayit Right(i,2) 'Part='PNum ' Desc='Description
    'GN DATABASE *ROOTSEG SSA1'
    Status=IMSQUERY('STATUS')
  End
  If Status='GB' then
    Call SayIt 'End of DataBase reached before' Segs 'records shown.'
  Else If Status~=' ' then Signal BadCall
  Call Sayit ' '
  Exit 0

SayIt: Procedure Expose IOPCB
  Parse Arg Msg
  'ISRT IOPCB MSG'
  If RC~=0 then Signal BadCall
Return

BadCall:
  'DLIINFO INFO'
  Parse Var Info Call PCB . . . Status .
  Msg = 'Unresolved Status Code' Status,
  'on' Call 'on PCB' PCB
  'ISRT IOPCB MSG'
Exit 99

```

Figure 64. PARTNUM Exec: Show Set of Parts Near a Specified Number

## PARTNAME Exec: Show a Set of Parts with a Similar Name

**Requirement:** The following REXX exec is designed to be run by the IVPREXX exec with PSB=DFSSAM02.

```

/* REXX EXEC TO SHOW ALL PARTS WITH A NAME CONTAINING A STRING */
/* Designed to be run by the IVPREXX exec with PSB=DFSSAM02 */
/* Syntax:  IVPREXX PARTNAME string <#parts> */

Arg PartName Segs .
Address REXXIMS
Term    ='IOPCB'      /* PCB Name */
DataBase='DBPCB01'   /* PCB Name for Parts Database */

Call SayIt 'IMS Parts DATABASE Transaction'
Call SayIt 'System Date:' Date('U') ' Time:' Time()
Call Sayit ' '

If ~DataType(Segs,'W') & Segs~='*' then Segs=5
If PartName='' then Do
    Call Sayit 'Please supply the first few characters of the part name'
    Exit 0
End

Call Sayit 'Request: Display' Segs 'Parts with Part Name like' PartName
SSA1='PARTRoot '
'GU DATABASE ROOT_SEG SSA1'
Status=REXXIMS('STATUS')
i=0
Do While RC=0 & (i<Segs | Segs~='*')
    Parse Var Root_Seg 3 PNum 18 27 Description 47
    'GN DATABASE ROOT_SEG SSA1'
    Status=REXXIMS('STATUS')
    If RC~=0 & Status~='GB' Then Leave
    If Index(Description,PartName)=0 then Iterate
    i=i+1
    Call Sayit Right(i,2)' Part='PNum ' Desc='Description
End
If RC~=0 & Status~='GB' Then Signal BadCall
If i<Segs & Segs~='*' then
    Call SayIt 'End of DataBase reached before' Segs 'records shown.'
Call Sayit ' '
Exit 0

SayIt: Procedure Expose Term
    Parse Arg Msg
    'ISRT Term MSG'
    If RC~=0 then Signal BadCall
Return

BadCall:
    Call "DFSSUT04" Term
Exit 99

```

Figure 65. PARTNAME Exec: Show Parts with Similar Names

## DFSSAM01 Exec: Load the Parts Database

For the latest version of the DFSSAM01 source code, see the IMS.ADFSEEXEC distribution library; member name is DFSSAM01.

## DOCMD: IMS Commands Front End

DOCMD is an automatic operator interface (AOI) transaction program that issues IMS commands and allows dynamic filtering of their output. The term “dynamic” means that you use the headers for the command as the selectors (variable names) in the filter expression (Boolean expression resulting in 1 if line is to be displayed and 0 if it is not). This listing is shown in Figure 72 on page 291.

Not all commands are allowed through transaction AOI, and some setup needs to be done to use this AOI.

**Related Reading:** See “Security Considerations for Automated Operator Commands” in *IMS Version 9: Administration Guide: System* for more information.

Some examples of DOCMD are given in Figure 66, Figure 67, Figure 68, Figure 69 on page 290, Figure 70 on page 290, and Figure 71 on page 290.

```
Please supply an IMS Command to execute.
EXEC DOCMD ended with RC= 0
```

Figure 66. Output from => DOCMD

```
Headers being shown for command: /DIS NODE ALL
Variable (header) #1 = RECTYPE
Variable (header) #2 = NODE_SUB
Variable (header) #3 = TYPE
Variable (header) #4 = CID
Variable (header) #5 = RECD
Variable (header) #6 = ENQCT
Variable (header) #7 = DEQCT
Variable (header) #8 = QCT
Variable (header) #9 = SENT
EXEC DOCMD ended with RC= 0
```

Figure 67. Output from => DOCMD /DIS NODE ALL;?

```
Selection criteria =>CID>0<= Command: /DIS NODE ALL
NODE_SUB TYPE  CID      RECD ENQCT DEQCT  QCT  SENT
L3270A  3277  01000004    5   19   19    0   26 IDLE CON
L3270C  3277  01000005  116  115  115    0  122 CON
Selected 2 lines from 396 lines.
DOCMD Executed 402 DL/I calls in 2.096787 seconds.
EXEC DOCMD ended with RC= 0
```

Figure 68. Output from => DOCMD /DIS NODE ALL;CID>0

```

Selection criteria =>TYPE=SLU2<= Command: /DIS NODE ALL
NODE_SUB TYPE CID RECD ENQCT DEQCT QCT SENT
WRIGHT SLU2 00000000 0 0 0 0 0 IDLE
Q3290A SLU2 00000000 0 0 0 0 0 IDLE
Q3290B SLU2 00000000 0 0 0 0 0 IDLE
Q3290C SLU2 00000000 0 0 0 0 0 IDLE
Q3290D SLU2 00000000 0 0 0 0 0 IDLE
V3290A SLU2 00000000 0 0 0 0 0 IDLE
V3290B SLU2 00000000 0 0 0 0 0 IDLE
H3290A SLU2 00000000 0 0 0 0 0 IDLE
H3290B SLU2 00000000 0 0 0 0 0 IDLE
E32701 SLU2 00000000 0 0 0 0 0 IDLE
E32702 SLU2 00000000 0 0 0 0 0 IDLE
E32703 SLU2 00000000 0 0 0 0 0 IDLE
E32704 SLU2 00000000 0 0 0 0 0 IDLE
E32705 SLU2 00000000 0 0 0 0 0 IDLE
ADLU2A SLU2 00000000 0 0 0 0 0 IDLE
ADLU2B SLU2 00000000 0 0 0 0 0 IDLE
ADLU2C SLU2 00000000 0 0 0 0 0 IDLE
ADLU2D SLU2 00000000 0 0 0 0 0 IDLE
ADLU2E SLU2 00000000 0 0 0 0 0 IDLE
ADLU2F SLU2 00000000 0 0 0 0 0 IDLE
ADLU2X SLU2 00000000 0 0 0 0 0 IDLE
ENDS01 SLU2 00000000 0 0 0 0 0 IDLE
ENDS02 SLU2 00000000 0 0 0 0 0 IDLE
ENDS03 SLU2 00000000 0 0 0 0 0 IDLE
ENDS04 SLU2 00000000 0 0 0 0 0 IDLE
ENDS05 SLU2 00000000 0 0 0 0 0 IDLE
ENDS06 SLU2 00000000 0 0 0 0 0 IDLE
NDSL2A1 SLU2 00000000 0 0 0 0 0 ASR IDLE
NDSL2A2 SLU2 00000000 0 0 0 0 0 ASR IDLE
NDSL2A3 SLU2 00000000 0 0 0 0 0 ASR IDLE
NDSL2A4 SLU2 00000000 0 0 0 0 0 ASR IDLE
NDSL2A5 SLU2 00000000 0 0 0 0 0 IDLE
NDSL2A6 SLU2 00000000 0 0 0 0 0 ASR IDLE
OMSSLU2A SLU2 00000000 0 0 0 0 0 IDLE
Selected 34 lines from 396 lines.
DOCMD Executed 435 DL/I calls in 1.602206 seconds.
EXEC DOCMD ended with RC= 0
    
```

Figure 69. Output from => DOCMD /DIS NODE ALL;TYPE=SLU2

```

Selection criteria =>ENQCT>0 & RECTYPE='T02'<= Command: /DIS TRAN ALL
TRAN CLS ENQCT QCT LCT PLCT CP NP LP SEGSZ SEGNO PARLM RC
TACP18 1 119 0 65535 65535 1 1 1 0 0 NONE 1
Selected 1 lines from 1104 lines.
DOCMD Executed 1152 DL/I calls in 5.780977 seconds.
EXEC DOCMD ended with RC= 0
    
```

Figure 70. Output from => DOCMD /DIS TRAN ALL;ENQCT>0 & RECTYPE='T02'

```

Selection criteria =>ENQCT>0<= Command: /DIS LTERM ALL
LTERM ENQCT DEQCT QCT
CTRL 19 19 0
T3270LC 119 119 0
Selected 2 lines from 678 lines.
DOCMD Executed 681 DL/I calls in 1.967670 seconds.
EXEC DOCMD ended with RC= 0
    
```

Figure 71. Output from => DOCMD /DIS LTERM ALL;ENQCT>0

The source code for the DOCMD exec is shown in Figure 72 on page 291.

```

/*****/
/* A REXX exec that executes an IMS command and parses the      */
/* output by a user supplied criteria.                          */
/*                                                              */
/*****/
/* Format:  tranname DOCMD IMS-Command;Expression              */
/* Where:   */
/*   tranname is the tranname of a command capable transaction that */
/*           will run the IVPREXX program.                       */
/*   IMS-Command is any valid IMS command that generates a table of */
/*           output like /DIS NODE ALL or /DIS TRAN ALL         */
/*   Expression is any valid REXX expression, using the header names*/
/*           as the variables, like CID>0 or SEND=0 or more     */
/*           complex like CID>0 & TYPE=SLU2                    */
/* Example: TACP18 DOCMD DIS A          Display active          */
/*           TACP18 DOCMD DIS NODE ALL;?   See headers of DIS NODE */
/*           TACP18 DOCMD DIS NODE ALL;CID>0 Show active Nodes  */
/*           TACP18 DOCMD DIS NODE ALL;CID>0 & TYPE='SLU2'     */
/*****/
Address REXXTDLI
Parse Upper Arg Cmd ';' Expression
Cmd=Strip(Cmd);
Expression=Strip(Expression)
If Cmd='' Then Do
  Call SayIt 'Please supply an IMS Command to execute.'
  Exit 0
End
AllOpt= (Expression='ALL')
If AllOpt then Expression='
If Left(Cmd,1)~='/' then Cmd='/'Cmd /* Add a slash if necessary */
If Expression='' Then
  Call SayIt 'No Expression supplied, all output shown',
  'from:' Cmd
Else If Expression='?' Then
  Call SayIt 'Headers being shown for command:' Cmd
Else
  Call SayIt 'Selection criteria =>'Expression'<=',
  'Command:' Cmd
x=Time('R'); Calls=0
ExitRC= ParseHeader(Cmd,Expression)
If ExitRC~=0 then Exit ExitRC
If Expression='?' Then Do
  Do i=1 to Vars.0
    Call SayIt 'Variable (header) #'i '=' Vars.i
    Calls=Calls+1
  End
End
End

```

Figure 72. DOCMD Exec: Process an IMS Command (Part 1 of 3)

```

Else Do
  Call ParseCmd Expression
  Do i=1 to Line.0
    If AllOpt then Line=Line.i
    Else Line=Substr(Line.i,5)
    Call SayIt Line
    Calls=Calls+1
  End
  If Expression=' ' then
    Call SayIt 'Selected' Line.0-1 'lines from',
      LinesAvail 'lines.'
  Else
    Call SayIt 'Total lines of output:' Line.0-1
    Call SayIt 'DOCMD Executed' Calls 'DL/I calls in',
      Time('E') 'seconds.'
  End
  Exit 0
ParseHeader:
  CurrCmd=Arg(1)
  CmdCnt=0
  'CMD IOPCB CURRCMD'
  CmdS= IMSQUERY('STATUS')
  Calls=Calls+1
  If CmdS=' ' then Do
    Call SayIt 'Command Executed, No output available.'
    Return 4
  End
  Else If CmdS='CC' then Do
    Call SayIt 'Error Executing Command, Status='CmdS
    Return 16
  End
  CurrCmd=Translate(CurrCmd,' ','15'x) /* Drop special characters */
  CurrCmd=Translate(CurrCmd,'_','-/') /* Drop special characters */
  CmdCnt=CmdCnt+1
  Interpret 'LINE.'||CmdCnt '= Strip(CurrCmd)'
  Parse Var CurrCmd RecType Header
  If Expression=' ' then Nop
  Else If Right(RecType,2)='70' then Do
    Vars.0=Words(Header)+1
    Vars.1 = "RECTYPE"
    Do i= 2 to Vars.0
      Interpret 'VARS.'i '= "'Word(CurrCmd,i)'"'
    End
  End
  Else Do
    Call SayIt 'Command did not produce a header',
      'record, first record's type='RecType
  End
  Return 12
End
Return 0

```

Figure 72. DOCMD Exec: Process an IMS Command (Part 2 of 3)



```

ParseCmd:
  LinesAvail=0
  CurrExp=Arg(1)
  Do Forever
    'GCMD IOPCB CURRCMD'
    CmdS= IMSQUERY('STATUS')
    Calls=Calls+1
    If CmdS=' ' then Leave
    /* Skip Time Stamps      */
    If Word(CurrCmd,1)='X99' & Expression=' ' then Iterate
    LinesAvail=LinesAvail+1
    CurrCmd=Translate(CurrCmd,' ','15'x)/* Drop special characters */
    If Expression=' ' then OK=1
    Else Do
      Do i= 1 to Vars.0
        Interpret Vars.i '= "'Word(CurrCmd,i)'"'
      End
      Interpret 'OK='Expression
    End
    If OK then Do
      CmdCnt=CmdCnt+1
      Interpret 'LINE.'||CmdCnt '= Strip(CurrCmd)'
    End
  End
  Line.0 = CmdCnt
  If CmdS='QD' Then
    Call SayIt 'Error Executing Command:',
      Arg(1) 'Stat='CmdS
  Return

SayIt: Procedure
  Parse Arg Line
  'ISRT IOPCB LINE'
  Return RC

```

Figure 72. DOCMD Exec: Process an IMS Command (Part 3 of 3)

---

## IVPREXX: MPP/IFP Front End for General Exec Execution

The IVPREXX exec is a front-end generic exec that is shipped with IMS as part of the IVP. It runs other execs by passing the exec name to execute after the TRANCODE (IVPREXX). For further details on IVPREXX, see “IVPREXX Sample Application” on page 265. For the latest version of the IVPREXX source code, see the IMS.ADFSEXEC distribution library; member name is IVPREXX.



---

## Part 3. Reference

<b>Chapter 15. Summary of DM and System Service Calls</b> . . . . .	297
Database Management Call Summary . . . . .	297
System Service Call Summary . . . . .	298
<b>Chapter 16. Command Codes Reference</b> . . . . .	301
<b>Chapter 17. CICS-DL/I User Interface Block Return Codes</b> . . . . .	303
Not-Open Conditions . . . . .	304
Invalid Request Conditions . . . . .	304



## Chapter 15. Summary of DM and System Service Calls

This chapter contains tables that summarize the database management and system service calls.

### In this Chapter:

- “Database Management Call Summary”
- “System Service Call Summary” on page 298

**Related Reading:** For detailed information on a specific call, see Chapter 4, “Writing DL/I Calls for Database Management,” on page 121 or Chapter 5, “Writing DL/I Calls for System Services,” on page 149. For information on the use of calls with programming language interfaces, see Chapter 3, “Defining Application Program Elements,” on page 77.

### Database Management Call Summary

Table 50 shows the parameters that are valid for each database management call. Optional parameters are enclosed in brackets ([ ]).

**Restriction:** Language-dependent parameters are not shown here. The variable *parmcount* is required for all PLITDLI calls. Either *parmcount* or **VL** is required for assembler language calls. Parmcount is optional in COBOL, C, and Pascal programs.

**Related Reading:** For more information on language-dependent application elements, see Chapter 3, “Defining Application Program Elements,” on page 77.

Table 50. Summary of DB Calls

Function Code	Meaning and Use	Options	Parameters	Valid for
CLSE	Close	Closes a GSAM database explicitly	function, gsam pcb or aib	DB/DC, DBCTL, DB batch, ODBA
DEQ <sup>b</sup>	Dequeue	Releases segments reserved by Q command code	function, i/o pcb (full function only), or aib, i/o area (full function only)	DB batch, BMP, MPP, IFP, DBCTL, ODBA
DLET	Delete	Removes a segment and its dependents from the database	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA
FLD <sup>b</sup>	Field	Accesses a field within a segment	function, db pcb or aib, i/o area, rootssa	DB/DC, ODBA
GHN <sup>b</sup>	Get Hold Next	Retrieves subsequent message segments	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA
GHNP	Get Hold Next in Parent	Retrieves dependents sequentially	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA
GHU <sup>b</sup>	Get Hold Unique	Retrieves segments and establishes a starting position in the database	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA
GN <sup>b</sup>	Get Next	Retrieves subsequent message segments	function, db pcb or aib, i/o area, [ssa or rsa]	DB/DC, DBCTL, DB batch, ODBA

Table 50. Summary of DB Calls (continued)

Function Code	Meaning and Use	Options	Parameters	Valid for
GNPb	Get Hold Next in Parent	Retrieves dependents sequentially	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA
GUbb	Get Unique	Retrieves segments and establishes a starting position in the database	function, db pcb or aib, i/o area, [ssa or rsa]	DB/DC, DBCTL, DB batch, ODBA
ISRT	Insert	Loads and adds one or more segments to the database	function, db pcb or aib, i/o area, [ssa or rsa]	DB/DC, DCCTL, DB batch, ODBA
OPEN	Open	Opens a GSAM database explicitly	function, gsam pcb or aib, [i/o area]	DB/DC, DBCTL, DB batch, ODBA
POSb	Position	Retrieves the location of a specific dependent or last-inserted sequential dependent segment	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA
REPL	Replace	Changes values of one or more fields in a segment	function, db pcb or aib, i/o area, [ssa]	DB/DC, DBCTL, DB batch, ODBA

## System Service Call Summary

Table 51 summarizes which system service calls you can use in each type of IMS DB application program and the parameters for each call. Optional parameters are enclosed in brackets ([ ]).

**Exception:** Language-dependent parameters are not shown here.

For more information on language-dependent application elements, see Chapter 3, "Defining Application Program Elements," on page 77.

Table 51. Summary of System Service Calls

Function Code	Meaning and Use	Options	Parameters	Valid for
CHKP (Basic)	Basic checkpoint; prepares for recovery	None	function, i/o pcb or aib, i/o area	DB batch, TM batch, BMP, MPP, IFP
CHKP (Symbolic)	Symbolic checkpoint; prepares for recovery	Specifies up to seven program areas to be saved	function, i/o pcb or aib, i/o area len, i/o area[, area len, area]	DB batch, TM batch, BMP
GMSG	Retrieves a message from the AO exit routine	Waits for an AOI message when none is available	function, aib, i/o area	DB/DC and DCCTL (BMP, MPP, IFP), DB/DC and DBCTL (DRA thread), DBCTL (BMP non-message driven), ODBA
GSCD <sup>1</sup>	Gets address of system contents directory	None	function, db pcb, i/o pcb or aib, i/o area	DB Batch, TM Batch

Table 51. Summary of System Service Calls (continued)

Function Code	Meaning and Use	Options	Parameters	Valid for
ICMD	Issues an IMS command and retrieves the first command response segment	None	function, aib, i/o area	DB/DC and DCCTL (BMP, MPP, IFP), DB/DC and DBCTL (DRA thread), DBCTL (BMP non-message driven), ODBA
INIT	Initialize; application receives data availability and deadlock occurrence status codes	Checks each PCB database for data availability	function, i/o pcb or aib, i/o area	DB batch, TM batch, BMP, MPP, IFP, DBCTL, ODBA
INQY	Inquiry; returns information and status codes about I/O or alternate PCB destination type, location, and session status	Checks each PCB database for data availability; returns information and status codes about the current execution environment	function, aib, i/o area, AIBFUNC=FINDI DBQUERYI ENVIRON	DB batch, TM batch, BMP, MPP, IFP, ODBA
LOGb	Log; writes a message to the system log	None	function, i/o pcb or aib, i/o area	DB batch, TM batch, BMP, MPP, IFP, DBCTL, ODBA
PCBb	Specifies and schedules another PSB	None	function, psb name, uibptr, [,sysserve]	CICS (DBCTL or DB/DC)
RCMD	Retrieves the second and subsequent command response segments resulting from an ICMD call	None	function, aib, i/o area	DB/DC and DCCTL (BMP, MPP, IFP), DB/DC and DBCTL (DRA thread), DBCTL (BMP non-message driven), ODBA
ROLB	Roll back; eliminates database updates	Returns last message to i/o area	function, i/o pcb or aib, i/o area	DB batch, TM batch, BMP, MPP, IFP
ROLL	Roll; eliminates database updates; abend	None	function	DB batch, TM batch, BMP, MPP, IFP
ROLS	Roll back to SETS; backs out database changes to SETS points	Issues call using name of DB PCB or i/o PCB	function, db pcb, i/o pcb or aib, i/o area, token	DB batch, TM batch, BMP, MPP, IFP, DBCTL, ODBA
SETS/SETU	Set a backout point; establishes as many as nine intermediate backout points	Cancels all existing backout points	function, i/o pcb or aib, i/o area, token	DB batch, TM batch, BMP, MPP, IFP, DBCTL, ODBA
SNAP <sup>2</sup>	Collects diagnostic information	Choose SNAP options	function, db pcb or aib, i/o area	DB batch, BMP, MPP, IFP, CICS (DBCTL or DB/DC), ODBA
STAT <sup>3</sup>	Statistics; retrieves IMS system statistics	Choose type and format	function, db pcb or aib, i/o area, stat function	DB batch, BMP, MPP, IFP, DBCTL, ODBA
SYNC	Synchronization; releases locked resources	Requests commit-point processing	function, i/o pcb or aib	BMP

Table 51. Summary of System Service Calls (continued)

Function Code	Meaning and Use	Options	Parameters	Valid for
TERM	Terminate; releases a PSB so another can be scheduled; commit database changes	None	function	CICS (DBCTL or DB/DC)
XRST	Extended restart; works with symbolic checkpoint to restart application program	Specifies up to seven areas to be saved	function, i/o pcb or aib, i/o area len, i/o area[, area len, area]	DB batch, TM batch, BMP

**Note:**

1. GSCD is a Product-sensitive programming interface.
2. SNAP is a Product-sensitive programming interface.
3. STAT is a Product-sensitive programming interface.



## Chapter 16. Command Codes Reference

This chapter contains the following reference information on all of the command codes:

- A brief description of each command code (see Table 52)
- A list of the calls you can use with each command code (see Table 53)

Table 52. Summary of Command Codes

Command Code	Allows You to...
C	Use the concatenated key of a segment to identify the segment.
D	Retrieve or insert a sequence of segments in a hierarchic path using only one call, instead of having to use a separate (path) call for each segment.
F	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	Retrieve the last occurrence of a segment under its parent.
M	Move a subset pointer to the next segment occurrence after your current position. (Used with DEDBs only.)
N	Designate segments that you do not want replaced when replacing segments after a Get Hold call. Usually used when replacing a path of segments.
P	Set parentage at a higher level than what it usually is (the lowest-level SSA of the call).
Q	Reserve a segment so that other programs will not be able to update it until you have finished processing and updating it.
R	Retrieve the first segment occurrence in a subset. (Used with DEDBs only.)
S	Unconditionally set a subset pointer to the current position. (Used with DEDBs only.)
U	Limit the search for a segment to the dependents of the segment occurrence on which position is established.
V	Use the current position at this hierarchic level and above as qualification for the segment.
W	Set a subset pointer to your current position, if the subset pointer is not already set. (Used with DEDBs only.)
Z	Set a subset pointer to 0, so it can be reused. (Used with DEDBs only.)
-	Null. Use an SSA in command code format without specifying the command code. Can be replaced during execution with the command codes that you want.

Table 53 shows the list of command codes with applicable calls.

Table 53. Command Codes and Calls

Command Code	GU	GHU	GN	GHN	GNP GHNP	REPL	ISRT	DLET
C		X		X	X		X	
D		X		X	X		X	
F		X		X	X		X	
L		X		X	X		X	

Table 53. Command Codes and Calls (continued)

Command Code	GU GHU	GN GHN	GNP GHNP	REPL	ISRT	DLET
M	X	X	X	X	X	
N				X		
P	X	X	X		X	
Q	X	X	X		X	
R	X	X	X		X	
S	X	X	X	X	X	
U	X	X	X		X	
V	X	X	X		X	
W	X	X	X	X	X	
Z	X	X	X	X	X	X
-	X	X	X	X	X	X

## Chapter 17. CICS-DL/I User Interface Block Return Codes

After issuing any kind of a DL/I call, CICS online programs must check the return code in the UIB before checking the DL/I status code. If the value in UIBRCODE is not null, the contents of the PCB status code are not meaningful.

For more information on defining and addressing a UIB, see “Specifying the UIB (CICS Online Programs Only)” on page 102.

The UIBRCODE contains two bytes, UIBFCTR and UIBDLTR. You should first check the contents of UIBFCTR; the contents of UIBDLTR are meaningful only if UIBFCTR indicates a NOTOPEN or INVREQ condition. Table 54,, Table 55,, and Table 56 show the return codes from the CICS-DL/I interface.

Table 54. Return Codes in UIBFCTR

Condition	ASM	COBOL	PL/I
NORESP (normal response)	X'00'	LOW-VALUES	00000 000
NOTOPEN (not open)	X'0C'	12-4-8-9	00001 100
INVREQ (invalid request)	X'08'	12-8-9	00001 000

Table 55. Return Codes in UIBDLTR if UIBFCTR='0C' (NOTOPEN)

Condition	ASM	COBOL	PL/I
Database not open	X'00'	LOW-VALUES	00000 000
Intent scheduling conflict	X'02'	12-2-9	00000 010

Table 56. Return Codes in UIBDLTR if UIBFCTR='08' (INVREQ)

Condition	ASM	COBOL	PL/I
Invalid argument passed to DL/I	X'00'	LOW-VALUES	00000 000
PSBNF (PSB not found)	X'01'	12-1-9	00000 001
PSBSCH (PSB already scheduled)	X'03'	12-3-9	00000 011
NOTDONE (request not executed)	X'04'	12-4-9	00000 100
PSBFAIL (PSB initialization failed)	X'05'	12-5-9	00000 101
TERMNS (termination not successful)	X'07'	12-7-9	00000 111
FUNCNS (function unscheduled)	X'08'	12-8-9	00001 000
INVPSB (invalid PSB)	X'10'	12-10-9	00010 000
DLINA (DL/I not active)	X'FF'	12-11-0-7-8-9	11111 111

If these codes do not appear to be due to programming errors, they may be caused by not-open or invalid-request conditions.

---

## Not-Open Conditions

A NOTOPEN condition is indicated if UIBFCTR contains X'0C'

---

### UIBDLTR='00'

**Explanation:** This is returned on a database call if the database was stopped after scheduling of the PSB.

using IMS program isolation.

---

### UIBDLTR='02'

**Explanation:** This indicates that an intent-scheduling conflict exists. This condition does not occur if you are

---

## Invalid Request Conditions

An invalid request is indicated by UIBFCTR=X'08'

---

### UIBDLTR='00' (INVARG)

**Explanation:** An invalid argument was passed to DL/I indicating one of these problems:

- Count argument exists, but count is too high.
- I/O area is missing.
- Received data length is greater than 65520.
- Call type is invalid.

- The master terminal operator has entered a RECOVERDB command. This command sets the do-not-schedule-flag in the DDIR. You will not be able to schedule any PSB that references the database.
- The END statement in the PDIR generation stream did not specify the DFSIDIR0 operand.

The trace entry, which contains the PCB status, gives you the reason for the scheduling failure.

---

### UIBDLTR='01' (PSBNF)

**Explanation:** This is returned after a scheduling call; it indicates that the PSB to be scheduled was not defined in the PSB directory (PDIR).

---

### UIBDLTR='07' (TERMNS)

**Explanation:** A terminate request was issued, but no PSB was currently scheduled. It could indicate that the PSB has already taken place because of a terminate request or CICS sync point.

---

### UIBDLTR='03' (PSBSCH)

**Explanation:** This PSB has already been scheduled.

---

### UIBDLTR='08' (FUNCNS)

**Explanation:** A database call was issued when the PSB was not scheduled.

---

### UIBDLTR='04' (NOTDONE)

**Explanation:** The XDLIPRE exit routine indicates that a DL/I request should not be issued.

---

### UIBDLTR='10' (INVPSB)

**Explanation:** SYSSERVE IOPCB specified for local DL/I.

---

### UIBDLTR='05' (PSBFAIL)

**Explanation:** The PSB could not be scheduled, possibly because:

- The database has been stopped.
- The master terminal operator has entered a DUMPDB command. This command sets the read-only flag in the DMB directory (DDIR). You will not be able to schedule any PSBs with update intent.

---

### UIBDLTR='FF' (DLINA)

**Explanation:** DLI=NO has been specified in the system initialization table (SIT).

---

## Part 4. Appendixes



---

## Appendix A. Sample Exit Routine (DFSREXXU)

IMS provides a sample user exit routine that is used with the IMS Adapter for REXX. For a description of how to write the user exit routine see *IMS Version 9: Customization Guide*. The sample user exit routine checks to see if it is being called on entry. If so, the user exit routine sets the parameter list to be the transaction code with no arguments and sets the start-up IMSRXTRC level to 2. The return code is set to 0. For the latest version of the DFSREXXU source code, see the IMS.SVSOURCE distribution library; member name is DFSREXXU.





## Appendix B. The DL/I Test Program (DFSDDLTO)

DFSDDLTO is an IMS application program test tool that issues calls to IMS based on control statement information. You can use it to verify and debug DL/I calls independently of application programs. You can run DFSDDLTO using any PSB, including those that use an IMS-supported language. You can also use DFSDDLTO as a general-purpose database utility program.

The functions that DFSDDLTO provides include:

- Issuing any valid DL/I call against any database using:
  - Any segment search argument (SSA) or PCB, or both
  - Any SSA or AIB, or both
- Comparing the results of a call to expected results. This includes the contents of selected PCB fields, the data returned in the I/O area, or both.
- Printing the control statements, the results of calls, and the results of comparisons only when the output is useful, such as after an unequal compare.
- Dumping DL/I control blocks, the I/O buffer pool, or the entire batch region.
- Punching selected control statements into an output file to create new test data sets. This simplifies the construction of new test cases.
- Merging multiple input data sets into a single input data set using a SYSIN2 DD statement in the JCL. You can specify the final order of the merged statements in columns 73 to 80 of the DFSDDLTO control statements.
- Sending messages to the z/OS system console (with or without waiting for a reply).
- Repeating each call up to 9,999 times.

### Control Statements

DFSDDLTO processes control statements to control the test environment. DFSDDLTO can issue calls to IMS full-function databases and Fast Path databases, as well as DC calls. Table 57 gives an alphabetical summary of the types of control statements DFSDDLTO uses. A detailed description of each type of statement follows.

Table 57. Summary of DFSDDLTO Control Statements

Control Statement	Code	Description
ABEND <sup>1</sup>	ABEND	Causes user abend 252.
CALL		There are two types of CALL statements:
	L	CALL FUNCTION identifies the type of IMS call function to be made and supplies information to be used by the call.
		CALL DATA provides IMS with additional information.
COMMENT		There are two types of COMMENT statements:
	T	Conditional allows a limited number of comments that are printed or not depending on how the STATUS statement is coded and the results of the PCB or DATA COMPARE.
	U <sup>1</sup>	Unconditional allows an unlimited number of comments, all of which are printed.
COMPARE		There are three types of COMPARE statements:

Table 57. Summary of DFSDDLTO Control Statements (continued)

Control Statement	Code	Description
	E	COMPARE DATA verifies that the correct segment was retrieved by comparing the segment returned by IMS with data in this statement.
		COMPARE AIB compares values that IMS returns to the AIB.
		COMPARE PCB checks fields in the PCB and calls for a snap dump of the DL/I blocks, the I/O buffer pool, or the batch region if the compare is unequal.
IGNORE	N or .	The program ignores statements that contain an N or . (period) in column 1.
OPTION <sup>1</sup>	O	Shows which control blocks are to be dumped, the number of unequal comparisons allowed, whether dumps are produced, number of lines printed per page, and the SPA size.
PUNCH <sup>1</sup>	CTL	PUNCH CTL produces an output data set consisting of the COMPARE PCB statements, the COMPARE AIB statements, the DATA statements, and all other control statements read.
STATUS <sup>1</sup>	S	Establishes print options and selects the PCB or AIB against which subsequent calls are to be issued.
WTO <sup>1</sup>	WTO	Sends a message to the z/OS console without waiting for reply.
WTOR <sup>1</sup>	WTOR	Sends a message to the z/OS console and waits for a reply before proceeding.

**Note:**

1. These control statements are acted on immediately when encountered in an input stream. Do not code them where they will interrupt call sequences. (See “Planning the Control Statement Order” on page 311.)

The control statements are further described below:

- The CALL statement is the central DFSDDLTO statement. The CALL statement has two parts: CALL FUNCTION and CALL DATA. CALL FUNCTION identifies the type of IMS call function and supplies information about segment search arguments (SSAs). CALL DATA provides more information required for the type of call identified by CALL FUNCTION.
- The STATUS statement controls the PCB, AIB, and handling of output.
- The three types of COMPARE statements, DATA, PCB, and AIB, compare different values:
  - If you want specific data from a call, use COMPARE DATA to check the segment data for mismatches when the call is made.
  - Use COMPARE PCB to check status codes, segment levels, and feedback keys. It also indicates mismatches when you specify output.
  - Use COMPARE AIB to compare values that IMS returns to the AIB.
- The two COMMENT statements, Conditional and Unconditional, allow you to set limits on the number of comments on the DFSDDLTO job stream and to specify whether you want the comments printed.
- The OPTION statement controls several overall functions such as the number of unequal comparisons allowed and the number of lines printed per page.
- The remaining statements, ABEND, IGNORE, CTL, WTO and WTOR, are not as important as the others at first. Read the sections describing these statements so that you can become familiar with the functions they offer.

When you are coding the DFSDDLTO control statements, keep the following items in mind:

- If you need to temporarily override certain control statements in the DFSDDLTO streams, go to the JCL requirements section and read about SYSIN/SYSIN2 processing under “SYSIN2 DD Statement” on page 348.
- You must fill in column 1 of each control statement. If column 1 is blank, the statement type defaults to the prior statement type. DFSDDLTO attempts to use any remaining characters as it would for the prior statement type.
- Use of reserved fields can produce invalid output and unpredictable results.
- Statement continuations are important, especially for the CALL statement.
- Sequence numbers are not required, but they can be very useful for some DFSDDLTO functions. To understand how to use sequence numbers, see “PUNCH Statement” on page 340, “SYSIN DD Statement” on page 347 and “SYSIN2 DD Statement” on page 348.
- All codes and fields in the DFSDDLTO statements must be left justified followed by blanks, unless otherwise specified.

---

## Planning the Control Statement Order

The order of control statements is *critical* in constructing a successful call. To avoid unpredictable results, follow these guidelines:

1. If you are using STATUS and OPTION statements, place them somewhere before the calls that are to use them.
2. Both types of COMMENT statements are optional but, if present, must appear before the call they document.
3. You must code CALL FUNCTION statements and any required SSAs consecutively without interruption.
4. CALL DATA statements must immediately follow the last continuation, if any, of the CALL FUNCTION statements.
5. COMPARE statements are optional but must follow the last CALL (FUNCTION or DATA) statement.
6. When CALL FUNCTION statements, CALL DATA statements, COMPARE DATA statements, COMPARE PCB statements, and COMPARE AIB statements are coded together, they form a call sequence. *Do not* interrupt call sequences with other DFSDDLTO control statements.  
**Exception:** IGNORE statements are the only exception to this rule.
7. Use IGNORE statements (N or .) to override any statement, regardless of its position in the input stream. You can use IGNORE statements in either SYSIN or SYSIN2 input streams.

---

## ABEND Statement

The ABEND statement causes IMS to issue an abend and terminate DFSDDLTO. Table 58 shows the format of the ABEND statement.

Table 58. ABEND Statement

Column	Function	Code	Description
1-5	Identifies control statement	ABEND	Issues abend U252. (No dump is produced unless you code DUMP on the OPTION statement.)
6-72	Reserved	b	
73-80	Sequence indication	nnnnnnnn	For SYSIN2 statement override.

## Examples of ABEND Statement

If you use ABEND in the input stream and want a dump, you must specify DUMP on the OPTION statement. The default on the OPTION statement is NODUMP.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
ABEND                                                                    22100010
```

Dump will be produced; OPTION statement provided requests dump.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
0 DUMP                                                                    22100010
```

No dump will be produced; OPTION statement provided requests NODUMP.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
0 NODUMP                                                                    22100010
```

---

## CALL Statement

The CALL control statement has two parts: CALL FUNCTION and CALL DATA.

- The CALL FUNCTION statement supplies the DL/I call function, the segment search arguments (SSAs), and the number of times to repeat the call. SSAs are coded according to IMS standards.
- The CALL DATA statement provides any data (database segments, z/OS commands, checkpoint IDs) required by the DL/I call specified in the CALL FUNCTION statement. See “CALL DATA Statement” on page 315.

## CALL FUNCTION Statement

Table 59 gives the format for CALL FUNCTION statements, including the column number, function, code, and description. This is the preferred format when you are not working with column-specific SSAs.

Table 59. CALL FUNCTION Statement

Column	Function	Code	Description
1	Identifies control statement	<b>L</b>	Issues an IMS call
2	Reserved	<b>b</b>	
3	SSA level	<b>b</b>	SSA level (optional)
		<b>n</b>	Range of hexadecimal characters allowed is 1-F
4	Reserved	<b>b</b>	
5-8	Repeat count	<b>bbbb</b>	If blank, repeat count defaults to 1.
		<b>nnnn</b>	'nnnn' is the number of times to repeat this call. Range is 1 to 9999, right-justified, with or without leading zeros.
9	Reserved	<b>b</b>	
10-13	Identifies DL/I call function	<b>bbbb</b>	If blank, use function from previous CALL statement.
		<b>xxxx</b>	'xxxx' is a DL/I call function.

Table 59. CALL FUNCTION Statement (continued)

Column	Function	Code	Description
	Continue SSA	CONT	Continuation indicator for SSAs too long for a single CALL FUNCTION statement. Column 72 of the preceding CALL FUNCTION statement must have an entry. The next CALL statement should have CONT in columns 10 - 13 and the SSA should continue in column 16.
14-15	Reserved	<b>b</b>	
16-23 or	SSA name	<b>xxxxxxxx</b>	Must be left-justified.
16-23 or	Token	<b>xxxxxxxx</b>	Token name (SETS/ROLS).
16-23 or	MOD name	<b>xxxxxxxx</b>	Modname (PURG+ISRT).
16-23 or	Subfunction	<b>xxxxxxxx</b>	nulls, DBQUERY, FIND, ENVIRON, PROGRAM (INQY).
16-19 and	Statistics type	<b>xxxx</b>	DBAS/DBES-OSAM or VBAS/VBES-VSAM (STAT). <sup>2</sup>
20 or	Statistics format	<b>x</b>	F - Formatted U- Unformatted S - Summary.
16-19	SETO ID <sup>1</sup>	<b>SETx</b>	Where x is 1, 2, or 3. Specified on SETO and CHNG calls as defined in Note.
21-24	SETO IOAREA SIZE	<b>nnnn</b>	Value of 0000 to 8192.  If a value greater than 8192 is specified, it defaults to 8192.  If no value is specified, the call is made with no SETO size specified.
24-71	Remainder of SSA		Unqualified SSAs must be blank. Qualified search arguments should have either an '*' or a '(' in column 24 and follow IMS SSA coding conventions.
72	Continuation column	<b>b</b>	No continuations for this statement.
		<b>x</b>	Alone, it indicates multiple SSAs each beginning in column 16 of successive statements. With CONT in columns 10-13 of the next statement, indicates a single SSA that is continued beginning in column 16 of the following statement.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

Table 59. CALL FUNCTION Statement (continued)

Column	Function	Code	Description
<b>Note:</b>			
1. SETO CALL:			
The SETO ID (SET1, SET2, or SET3) is required on the SETO call if DFSDDLTO is to keep track of the text unit address returned on the SETO call that would be passed on the CHNG call for option parameter TXTU.			
If the SETO ID is omitted on the SETO call, DFSDDLTO does not keep track of the data returned and is unable to reference it on a CHNG call.			
CHNG CALL:			
The SETO ID (SET1, SET2, or SET3) is required on the CHNG call if DFSDDLTO is to place the address of the SETO ID I/O area returned on the SETO call. This is the SETO call of the text unit returned on the SETO call with a matching SETO ID for this CHNG call into the "TXTU=ADDR" field of the option parameter in the CHNG call.			
When the SETO ID is specified on the CHNG call, DFSDDLTO moves the address of that text unit returned on the SETO call using the same SETO ID.			
Code the OPTION statement parameter TXTU as follows: TXTU=xxxx where xxxx is any valid non-blank character. It cannot be a single quote character.			
Suggested value for xxxx could be SET1, SET2, or SET3. This value is not used by DFSDDLTO.			
2. STAT is a Product-sensitive programming interface.			

The following information applies to different types of continuations:

- Column 3, the SSA level, is usually blank. If it is blank, the first CALL FUNCTION statement fills SSA 1, and each following CALL FUNCTION statement fills the next lower SSA. If column 3 is not blank, the statement fills the SSA at that level, and the following CALL FUNCTION statement fills the next lower one.
  - Columns 5 through 8 are usually blank, but if used, must be right justified. The same call is repeated as specified by the repeat call function.
  - Columns 10 through 13 contain the DL/I call function. The call function is required only for the first CALL FUNCTION statement when multiple SSAs are in a call. If left blank, the call function from the previous CALL FUNCTION statement is used.
  - Columns 16 through 23 contain the segment name if the call uses an SSA.
  - If the DL/I call contains multiple SSAs, the statement must have a nonblank character in column 72, and the next SSA must start in column 16 of the next statement. The data in columns 1 and 10 through 13 are blank for the second through last SSAs.
- Restriction:** On ISRT calls, the last SSA can have only the segment name with no qualification or continuation.
- If a field value extends past column 71, put a nonblank character in column 72. (This character is not read as part of the field value, only as a continuation character.) In the next statement insert the keyword CONT in columns 10 through 13 and continue the field value starting at column 16.
  - Maximum length for the field value is 256 bytes, maximum size for an SSA is 290 bytes, and the maximum number of SSAs for this program is 15, which is the same as the IMS limit.
  - If columns 5 through 8 in the CALL FUNCTION statement contain a repeat count for the call, the call will terminate when reaching that count, unless it first encounters a GB status code.

**Related Reading:** See "CALL FUNCTION Statement with Column-Specific SSAs" on page 329 for another format supported by DFSDDLTO.

## CALL DATA Statement

CALL DATA statements provide IMS with information normally supplied in the I/O area for that type of call function.

CALL DATA statements must follow the last CALL FUNCTION statement. You must enter an L in column 1, the keyword DATA in columns 10 through 13, and code the necessary data in columns 16 through 71. You can continue data by entering a nonblank character in column 72. On the continuation statement, columns 1 through 15 are blank and the data resumes in column 16. Table 60 shows the format for a CALL DATA statement.

Table 60. CALL DATA Statement

Column	Function	Code	Description
1	Identifies control statement	L	CALL DATA statement.
2	Increase segment length	K	Adds 2500 bytes to the length of data defined in columns 5 through 8.
3	Propagate remaining I/O indicator	P	Causes 50 bytes (columns 16 through 65) to be propagated through remaining I/O area. <b>Note:</b> This must be the last data statement and cannot be continued.
4	Format options	b	Not a variable-length segment.
		V	For the first statement describing the only variable-length segment or the first variable-length segment of multiple variable-length segments, LL field is added before the segment data.
		M	For statements describing the second through the last variable-length segments, LL field is added before the segment data.
		P	For the first statement describing a fixed-length segment in a path call.
		Z	For message segment, LLZZ field is added before the data.
		U	Undefined record format for GSAM records. The length of segment for an ISRT is placed in the DB PCB key feedback area.
5-8	Length of data in segment	nnnn	This value must be right justified but need not contain leading zeros. If you do not specify a length, DFSDDLTO will use the number of DATA statements read multiplied by 56 to derive the length.
9	Reserved	b	
10-13	Identifies CALL DATA statement	DATA	Identifies this as a DATA statement.
14-15	Reserved	b	
16-71	Data area	xxxx	Data that goes in the I/O area.
or			
16-23	Checkpoint ID		Checkpoint ID (SYNC).
or			

Table 60. CALL DATA Statement (continued)

Column	Function	Code	Description
16-23	Destination name		Destination name (CHNG).
or			
16	DEQ option		DEQ options (A,B,C,D,E,F,G,H,I, or J).
72	Continuation column	<b>b</b>	If no more continuations for this segment.
		<b>x</b>	If more data for this segment or more segments.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

When inserting variable-length segments or including variable-length data for a CHKP or LOG call:

- You must use a V or M in column 4 of the CALL DATA statement.
- Use V if only one variable-length segment is being processed.
- You must enter the length of the data with leading zeros, right justified, in columns 5 through 8. The value is converted to binary and becomes the first 2 bytes of the segment data.
- You can continue a CALL DATA statement into the next CALL DATA statement by entering a nonblank character in column 72. For subsequent statements, leave columns 1 through 15 blank, and start the data in column 16.

If multiple variable-length segments are required (that is, concatenated logical child/logical parent segments, both of which are variable-length) for the first segment:

- You must enter a V in column 4.
- You must enter the length of the first segment in columns 5 through 8.
- If the first segment is longer than 56 bytes, continue the data as described for inserting variable-length segments.

**Exceptions:**

- The last CALL DATA statement to contain data for this segment must have a nonblank character in column 72.
- The next CALL DATA statement applies to the next variable-length statement and must contain an M in column 4 and the length of the segment in columns 5 through 8.

You can concatenate any number of variable-length segments in this manner. Enter M or V and the length (only in CALL DATA statements that begin data for a variable-length segment).

When a program is inserting or replacing through path calls:

- Enter a P in column 4 to specify that the length field is to be used as the length the segment will occupy in the user I/O area.
- You only need to use P in the first statement of fixed-length-segment CALL DATA statements in path calls that contain both variable- and fixed-length segments.
- You can use V, M, and P in successive CALL DATA statements.

For INIT, SETS, ROLS, and LOG calls:

- The format of the I/O area is  
LLZZuser-data



where LL is the length of the data in the I/O area, including the length of the LLZZ portion.

- If you want the program to use this format for the I/O area, enter a Z in column 4 and the length of the data in columns 5 through 8. The length in columns 5 through 8 is the length of the data, not including the 4-byte length of LLZZ.

## OPTION DATA Statement

The OPTION DATA statement contains options as required for SETO and CHNG calls.

Table 61 shows the format for an OPTION DATA statement, including the column number, function, code, and description.

Table 61. OPTION DATA Statement

Column	Function	Code	Description
1	Identifies control statement	L	OPTION statement.
2-9	Reserved	b	
10-13	Identifies	OPT	Identifies this as OPTION statement.
		CONT	Identifies this as a continuation of an option input.
14-15	Reserved	b	
16-71	Option area	xxxx	Options as defined for SETO and CHNG call.
72	Continuation column	b	If no more continuations for options.
		x	If more option data exists in following statement.
73-80	Sequence number	nnnnnnnn	For SYSIN2 statement override.

## FEEDBACK DATA Statement

The FEEDBACK DATA statement defines an area to contain feedback data.

The FEEDBACK DATA statement is optional. However, if the FEEDBACK DATA statement is used, an OPTION DATA statement is required.

Table 62 shows the format for a FEEDBACK DATA statement, including the column number, function, code, and description.

Table 62. FEEDBACK DATA Statement

Column	Function	Code	Description
1	Identifies control statement	L	FEEDBACK statement.
2-3	Reserved	b	
4	Format option	b	Feedback area contains LLZZ.
		Z	Length of feedback area will be computed and the LLZZ will be added to the feedback area.
5-8	Length of feedback area	nnnn	This value must be right justified but need not contain leading zeros. If you do not specify a length, DFSDDLTO uses the number of FDBK inputs read multiplied by 56 to derive the length.
2-9	Reserved	b	

Table 62. FEEDBACK DATA Statement (continued)

Column	Function	Code	Description
10-13	Identifies	<b>FDBK</b>	Identifies this as feedback statement and continuation of feedback statement.
14-15	Reserved	<b>b</b>	
16-71	Feedback area	<b>xxxx</b>	Contains user pre-defined initialized area.
72	Continuation column	<b>b</b>	If no more continuations for feedback.
		<b>x</b>	If more feedback data exists in following statement.
73-80	Sequence number	<b>nnnnnnnn</b>	For SYSIN2 statement override.

## Call Functions

### DL/I Call Functions

Table 63 shows the DL/I call functions supported in DFSDDLTO and which ones require data statements.

Table 63. DL/I Call Functions

Call	AIB Support	PCB Support	Data Stmt <sup>1</sup>	Description
<b>CHKP</b>	yes	yes	R	Checkpoint.
<b>CHNG</b>	yes	yes	R	Change alternate PCB.
			R	Contains the alternate PCB name option statement and feedback statement optional.
<b>CMD</b>	yes	yes	R	Issue IMS command. This call defaults to I/O PCB.
<b>DEQ</b>	yes	yes	R	Dequeue segments locked with the Q command code. For full function, this call defaults to the I/O PCB, provided a DATA statement containing the class to dequeue immediately follows the call. For Fast Path, the call is issued against a DEDB PCB. Do not include a DATA statement immediately following the DEQ call.
<b>DLET</b>	yes	yes	O	Delete. If the data statement is present, it is used. If not, the call uses the data from the previous Get Hold Unique (GHU).
<b>FLD</b>	yes	yes	R	Field—for Fast Path MSDB calls using FSAs. This call references MSDBs only. If there is more than one FSA, put a nonblank character in column 34, and put the next FSA in columns 16-34 of the next statement. A DATA statement containing FSA is required.
<b>GCMD</b>	yes	yes	N	Get command response. This call defaults to I/O PCB.
<b>GHN</b>	yes	yes	O <sup>2</sup>	Get Hold Next.
<b>GHNP</b>	yes	yes	O <sup>2</sup>	Get Hold Next in Parent.
<b>GHU</b>	yes	yes	O <sup>2</sup>	Get Hold Unique.
<b>GMSG</b> <sup>3</sup>	yes	no	R	Get Message is used in an automated operator (AO) application program to retrieve a message from AO exit routine DFSAOE00. The DATA statement is required to allow for area in which to return data. The area must be large enough to hold this returned data.
<b>GN</b>	yes	yes	O <sup>2</sup>	Get Next segment.
<b>GNP</b>	yes	yes	O <sup>2</sup>	Get Next in Parent.

Table 63. DL/I Call Functions (continued)

Call	AIB Support	PCB Support	Data Stmt <sup>1</sup>	Description
<b>GU</b>	yes	yes	O <sup>2</sup>	Get Unique segment.
<b>ICMD<sup>3</sup></b>	yes	no	R	Issue Command enables an automated operator (AO) application program to issue an IMS command and retrieve the first command response segment. The DATA statement is required to contain the input command and to allow for area in which to return data. The area must be large enough to hold this returned data.
<b>INIT</b>	yes	yes	R	Initialization This call defaults to I/O PCB. A DATA statement is required. Use the LLZZ format.
<b>INQY<sup>3</sup></b>	yes	no	R	Request environment information using the AIB and the ENVIRON subfunction. The DATA statement is required to allow for area in which to return data. The area must be large enough to hold this returned data.
			R	Request database information using the AIB and the DBQUERY subfunction, which is equivalent to the INIT DBQUERY call. The DATA statement is required to allow for area in which to return data. The area must be large enough to hold this returned data.
<b>ISRT</b>	yes	yes		Insert.
			R	DB PCB, DATA statement required.
			O	I/O PCB using I/O area with MOD name, if any, in columns 16-23.
			R	Alt PCB.
<b>LOG</b>	yes	yes	R	Log system request. This call defaults to I/O PCB. DATA statement is required and can be specified in one of two ways.
<b>POS</b>	yes	yes	N	Position - for DEDBs to determine a segment location. This call references DEDBs only.
<b>PURG</b>	yes	yes		Purge.
			R	This call defaults to use I/O PCB. If column 16 is not blank, MOD (message output descriptor) name is used and a DATA statement is required.
			O	If column 16 is blank, the DATA statement is optional.
<b>RCMD<sup>3</sup></b>	yes	no	R	Retrieve Command enables an automated operator (AO) application program to retrieve the second and subsequent command response segments after an ICMD call. The DATA statement is required to allow for area in which to return data. The area must be large enough to hold this returned data.
<b>REPL</b>	yes	yes	R	Replace—This call references DB PCBs only. The DATA statement is required.
<b>ROLB</b>	yes	yes	O	Roll Back call.
<b>ROLL</b>	no	yes	O	Roll Back call and issue U778 abend.

Table 63. DL/I Call Functions (continued)

Call	AIB Support	PCB Support	Data Stmt <sup>1</sup>	Description
<b>ROLS</b>	yes	yes	O	Back out updates and issue 3303 abend. Uses the I/O PCB. Can be used with the SETS call function. To issue a ROLS with an I/O area and token as the fourth parameter, specify the 4-byte token in column 16 of the CALL statement. Leaving columns 16-19 blank will cause the call to be made without the I/O area and the token. (To issue a ROLS using the current DB PCB, use ROLX.)
<b>ROLX</b>	yes	yes	O	Roll call against the DB PCB (DFSDDLTO call function). This call is used to request a Roll Back call to DB PCB, and is changed to ROLS call when making the DL/I call.
<b>SETO</b>	yes	yes	N	Set options. OPTION statement required. FEEDBACK statement optional.
<b>SETS/SETU</b>	yes	yes	O	Create or cancel intermediate backout points. Uses I/O PCB. To issue a SETS with an I/O area and token as the fourth parameter, specify the four-byte token in column 16 of the CALL statement and include a DATA statement. Leaving columns 16-19 blank will cause the call to be made without the I/O area and the token.
<b>SNAP<sup>4</sup></b>	yes	yes	O	Sets the identification and destination for snap dumps. If a SNAP call is issued without a CALL DATA statement, a snap of the I/O buffer pools and control blocks will be taken and sent to LOG if online and to PRINTDD DCB if batch. The SNAP ID will default to SNAPxxxx where xxxx starts at 0000 and is incremented by 1 for every SNAP call without a DATA statement. The SNAP options default to YYYYN. If a CALL DATA statement is used, columns 16-23 specify the SNAP destination, columns 24-31 specify the SNAP identification, and columns 32-35 specify the SNAP options. SNAP options are coded using 'Y' to request a snap dump and 'N' to prevent it. Column 32 snaps the I/O buffer pools, columns 33 and 34 snap the IMS control blocks and column 35 snaps the entire region. The SNAP call function is only supported for full-function database PCB.
<b>STAT<sup>5</sup></b>	yes	yes	O	The STAT call retrieves statistics on the IMS system. This call must reference only full-function DB PCBs. See the examples on 329. Statistics type is coded in columns 16-19 of the CALL FUNCTION statement.  <b>DBAS</b> For OSAM database buffer pool statistics.  <b>VBAS</b> For VSAM database subpool statistics. Statistics format is coded in column 20 of the CALL FUNCTION statement.  <b>F</b> For the full statistics to be formatted if F is specified, the I/O area must be at least 360 bytes.  <b>U</b> For the full statistics to be unformatted if U is specified, the I/O area must be at least 72 bytes.  <b>S</b> For a summary of the statistics to be formatted if S is specified, the I/O area must be at least 120 bytes.
<b>SYNC</b>	yes	yes	R	Synchronization.
<b>XRST</b>	yes	yes	R	Restart.

Table 63. DL/I Call Functions (continued)

Call	AIB Support	PCB Support	Data Stmt <sup>1</sup>	Description
------	-------------	-------------	------------------------	-------------

**Notes:**

1. R = required; O = optional; N = none
2. The data statement is required on the AIB interface.
3. Valid only on the AIB interface.
4. SNAP is a Product-sensitive programming interface.
5. STAT is a Product-sensitive programming interface.

## Examples of DL/I Call Functions

**Basic CHKP Call:** Use a CALL FUNCTION statement to contain the CHKP function and a CALL DATA statement to contain the checkpoint ID.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      CHKP                                     10101400
L      DATA TESTCKPT
```

**Symbolic CHKP Call with Two Data Areas to Checkpoint:** Use a CALL FUNCTION statement to contain the CHKP function, a CALL DATA statement to contain the checkpoint ID data, and two CALL DATA statements to contain the data that you want to checkpoint.

You also need to use an XRST call when you use the symbolic CHKP call. Prior usage of an XRST call is required when using the symbolic CHKP call, as the CHKP call keys on the XRST call for symbolic CHKP.

**Recommendation:** Issue an XRST call as the first call in the application program.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      XRST
L      .
L      .
L      .
L      CHKP
L      DATA TSTCHKP2                               X
L      8 DATA STRING2-                             X
L      16 DATA STRING2-STRING2-
U EIGHT BYTES OF DATA (STRING2-) IS CHECKPOINTED AND
U SIXTEEN BYTES OF DATA (STRING2-STRING2-) IS CHECKPOINTED ALSO
```

**CHNG Call:** Use a CALL FUNCTION statement to contain the CHNG function and a CALL DATA statement to contain the new logical terminal name.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      CHNG SET1
L      OPT IAFP=A1M,PRTO=LLOPTION1,OPTION2,
L      CONT OPTION4
L Z0023 DATA DESTNAME
```

LL is the hex value of the length of LLOPTION,.....OPTION4.

The following is an example of a CHNG statement using SETO ID SET2, OPTION statement, DATA statement with MODNAME, and FDBK statement.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      CHNG SET2
L      OPT  IAFP=A1M,XTU=SET2
L Z0023  DATA DESTNAME
L Z0095  FDBK FEEDBACK AREA

```

**CMD Call:** Use a CALL FUNCTION statement to contain the CMD function and a CALL DATA statement to contain the Command data.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      CMD
L ZXXXX DATA  COMMAND DATA

```

WHERE XXXX = THE LENGTH OF THE COMMAND DATA

**DEQ Call:** For full function, use a CALL FUNCTION statement to contain the DEQ function and a CALL DATA statement to contain the DEQ value (A,B,C,D,E,F,G,H,I or J).

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      DEQ
L      DATA  A

```

For Fast Path, use a CALL FUNCTION statement to contain the DEQ function.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      DEQ

```

**DLET Call:** Use a CALL FUNCTION statement to contain the DLET function. The data statement is optional. If there are intervening calls to other PCBs between the Get Hold call and the DLET call, you must use a data statement to refresh the I/O area with the segment to be deleted.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      DLET

```

**FLD Call:** Use a CALL FUNCTION statement to contain the FLD function and ROOTSSA, and a CALL DATA statement to contain the FSAs.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      FLD  ROOTA  (KEYA  =ROOTA)
L      DATA  ??????
L      DATA

```

**GCMD Call:** Use a CALL FUNCTION statement to contain the GCMD function; no CALL DATA statement is required.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      GCMD

```

**GHN Call:** Use a CALL FUNCTION statement to contain the GHN function; no CALL DATA statement is required.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      GHN

```

**GHNP Call:** Use a CALL FUNCTION statement to contain the GHNP function; no CALL DATA statement is required.



```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      GU   SEGA   *D(KEYA   = A200)                               *
          SEGF   *D(KEYF   = F250)                               *
          SEGG   *D(KEYG   = G251)

```

**ICMD Call:** Use a CALL FUNCTION statement to contain the ICMD function. Use a CALL DATA statement to contain the command.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      ICMD
L Z0132 DATA /DIS ACTIVE

```

**INIT Call:** Use a CALL FUNCTION statement to contain the INIT call and a CALL DATA statement to contain the INIT function DBQUERY, STATUS GROUPA, or STATUS GROUPB.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      INIT                               10103210
L Z0011 DATA DBQUERY

```

**INQY Call:** Use a CALL FUNCTION statement to contain the INQY call and either the DBQUERY or ENVIRON subfunction. The subfunctions are in the call input rather than the data input as in the INIT call.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      INQY ENVIRON                               10103210
L V0256 DATA                               10103211
L                                           10103212

```

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      INQY DBQUERY                               10103210
L V0088 DATA                               10103211
L                                           10103212

```

**ISRT Call:** Use two CALL FUNCTION statements to contain the multiple SSAs and a CALL DATA statement to contain the segment data.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      ISRT STOCKSEG(NUMFIELD =20011)           X10103210
          ITEMSSEG                               10103211
L V0018 DATA 3002222222222222                10103212

```

**ISRT Containing Only One Fixed-Length Segment:** Use a CALL FUNCTION statement to contain the ISRT function and segment name, and two CALL DATA statements to contain the fixed-length segment. When inserting only one fixed-length segment, leave columns 4 through 8 blank and put data in columns 16 through 71. To continue data, put a nonblank character in column 72, and the continued data in columns 16 through 71 of the next statement.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      ISRT JOKESSEG                               10103210
L      DATA THEQUICKBLACKDOGJUMPEDONTOTHECRAZYFOXOOPSTHEQUICKBROWNFO*10103211
          XJUMPEDOVERTHELAZYDOGSIR             10103212

```

**ISRT Containing Only One Variable-Length Segment:** Use a CALL FUNCTION statement to contain the ISRT function and segment name, and two CALL DATA statements to contain the variable-length segment. When only one segment of variable-length is being processed, you must enter a V in column 4, and columns 5 through 8 must contain the length of the segment data. The length in columns 5 through 8 is converted to binary and becomes the first two bytes of the segment



data. To continue data, put a nonblank character in column 72, and the continued data in columns 16 through 71 of the next statement.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      ISRT  JOKESSEG                                10103210
L  V0080 DATA  THEQUICKBLACKDOGJUMPEDONTOTHECRAZYFOXOOPSTHEQUICKBROWNFO*10103211
                                XJUMPEDOVERTHELAZYDOGSIR                                10103212
```

**ISRT Containing Multiple Variable-Length Segments:** Use a CALL FUNCTION statement to contain the ISRT function and segment name, and four CALL DATA statements to contain the variable-length segments. For the first segment, you must enter a V in column 4 and the length of the segment data in columns 5 through 8. If the segment is longer than 56 bytes, put a nonblank character in column 72, and continue data on the next statement as described above. The last statement to contain data for this segment must have a nonblank character in column 72.

The next DATA statement applies to the next variable-length segment and it must contain an M in column 4, the length of the new segment in columns 5 through 8, and data starting in column 16. Any number of variable-length segments can be concatenated in this manner. If column 72 is blank, the next statement must have the following:

- An L in column 1
- An M in column 4
- The length of the new segment in columns 5 through 8
- The keyword DATA in columns 10 through 13
- Data starting in column 16

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      ISRT  AAAASEG                                10103210
L  V0080 DATA  THEQUICKBLACKDOGJUMPEDONTOTHECRAZYFOXOOPSTHEQUICKBROWNFO*10103211
                                XJUMPEDOVERTHELAZYDOGSIR                                *10103212
M0107 DATA  NOWISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOFTHEIRCOUNTRYNOW*10103213
                                ISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOFTHEIRCOUNTRY      10103214
```

**ISRT Containing Multiple Segments in a PATH CALL:** Use a CALL FUNCTION statement to contain the ISRT function and segment name, and seven CALL DATA statements to contain the multiple segments in the PATH CALL.

When DFSDDLTO is inserting or replacing segments through path calls, you can use V and P in successive statements. The same rules apply for coding multiple variable-length segments, but fixed-length segments must have a P in column 4 of the DATA statement. This causes the length field in columns 5 through 8 to be used as the length of the segment, and causes the data to be concatenated in the I/O area without including the LL field.

Rules for continuing data in the same segment or starting a new segment in the next statement are the same as those applied to the variable-length segment.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      ISRT  LEV01SEG*D                               *10103210
          LEV02SEG                                   *10103211
          LEV03SEG                                   *10103212
          LEV04SEG                                   10103213
L  V0080 DATA THEQUICKBLACKDOGJUMPEDONTOTHECRAZYFOXOOPSTHEQUICKBROWNFO*10103214
          XJUMPEDOVERTHELAZYDOGSIR                 *10103215
          M0107 DATA NOWISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOFTHEIRCOUNTRYNOW*10103216
          ISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOFTHEIRCOUNTRY *10103217
L  P0039 DATA THEQUICKBROWNFOXJUMPEDOVERTHELAZYDOGSIR *10103218
L  M0107 DATA NOWISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOFTHEIRCOUNTRYNOW*10103219
          ISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOFTHEIRCOUNTRY 10103220

```

**LOG Call Using an LLZZ Format:** Use a CALL FUNCTION statement to contain the LOG function and a CALL DATA statement to contain the LLZZ format of data to be logged.

When you put a Z in column 4, the first word of the record is not coded in the DATA statement. The length specified in columns 5 through 8 must include the 4 bytes for the LLZZ field that is not in the DATA statement.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      LOG                                           10103210
L  Z0016 DATA ASEGMENT ONE                          10103211

```

The A in column 16 becomes the log record ID.

**POS Call:** Use a CALL FUNCTION statement to contain the POS function and SSA; CALL DATA statement is optional.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      POS  SEGA  (KEYA  =A300)

```

**PURG Call with MODNAME and Data:** Use a CALL FUNCTION statement to contain the PURG function and MOD name. Use the CALL DATA statement to contain the message data. If MOD name is provided, a DATA statement is required.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      PURG MODNAME1
L      DATA FIRST SEGMENT OF NEW MESSAGE

```

**PURG Call with Data and no MODNAME:** Use a CALL FUNCTION statement to contain the PURG function; a DATA statement is optional.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      PURG
L      DATA FIRST SEGMENT OF NEW MESSAGE

```

**PURG Call without MODNAME or Data:** Use a CALL FUNCTION statement to contain the PURG function; CALL DATA statement is optional.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      PURG

```

**RCMD Call:** Use a CALL FUNCTION statement to contain the RCMD function. Use a CALL DATA statement to retrieve second and subsequent command response segments resulting from an ICMD call.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          RCMD
L  Z0132 DATA
```

**REPL Call:** Use a CALL FUNCTION statement to contain the REPL function. Use a CALL DATA statement to contain the replacement data.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          REPL
L  V0028 DATA THIS IS THE REPLACEMENT DATA
```

**ROLB Call Requesting Return of First Segment of Current Message:** Use a CALL FUNCTION statement to contain the ROLB function. Use the CALL DATA statement to request first segment of current message.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          ROLB
L          DATA THIS WILL BE OVERLAID WITH FIRST SEGMENT OF MESSAGE
```

**ROLB Call Not Requesting Return of First Segment of Current Message:** Use a CALL FUNCTION statement to contain the ROLB function. The CALL DATA statement is optional.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          ROLB
```

**ROLL Call:** Use a CALL FUNCTION statement to contain the ROLL function. The CALL DATA statement is optional.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          ROLL
```

**ROLS Call with a Token:** Use a CALL FUNCTION statement to contain the ROLS function and token, and the CALL DATA statement to provide the data area that will be overlaid by the data from the SETS call.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          ROLS TOKEN1

L  Z0046 DATA THIS WILL BE OVERLAID WITH DATA FROM SETS
```

**ROLS Call without a Token:** Use a CALL FUNCTION statement to contain the ROLS function. The CALL DATA statement is optional.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          ROLS
```

**ROLX Call:** Use a CALL FUNCTION statement to contain the ROLX function. The CALL DATA statement is optional. The ROLX function is treated as a ROLS call with no token.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          ROLX
```

**SETO Call:** Use a CALL FUNCTION statement to contain the SETO function. The DATA statement is optional; however, if an OPTION statement is passed on the call, the DATA statement is required. Also, if a FEEDBACK statement is passed on the call, then both the DATA and OPTION statements are required. The following is an

example of a SETO statement using the OPTION statement and SETO token of SET1.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      SETO  SET1 5000
L      OPT  PRT0=11OPTION1,OPTION2,
L      CONT OPTION3,
L      CONT OPTION4
```

11 is the hex value of the length of 11OPTION,.....OPTION4.

The following is an example of a SETO statement using the OPTION statement and SETO token of SET1.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      SETO  SET1 7000
L      OPT  PRT0=11OPTION1,OPTION2,OPTION3,OPTION4
```

11 is the hex value of the length of 11OPTION,.....OPTION4.

The following is an example of a SETO statement using the OPTION statement and SETO token of SET2 and FDBK statement.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      SETO  SET2 5500
L      OPT  PRT0=11OPTION1,OPTION2,OPTION3,OPTION4
L Z0099  FDBK OPTION ERROR FEEDBACK AREA
```

11 is the hex value of the length of 11OPTION,.....OPTION4.

**SETS Call with a Token:** Use a CALL FUNCTION statement to contain the SETS function and token; use the CALL DATA statement to provide the data that is to be returned to ROLS call.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      SETS  TOKEN1

L Z0033 DATA  RETURN THIS DATA ON THE ROLS CALL
```

**SETS Call without a Token:** Use a CALL FUNCTION statement to contain the SETS function; CALL DATA statement is optional.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      SETS
```

This section (SNAP call) contains product-sensitive programming interface information.

**SNAP Call:** Use a CALL FUNCTION statement to contain the SNAP function and a CALL DATA statement to contain the SNAP data.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L      SNAP                                     10103210
L V0022 DATA  PRINTDD 2222222                10103212
```

This section (STAT call) contains product-sensitive programming interface information.

**STAT Call:** OSAM statistics require only one STAT call. STAT calls for VSAM statistics retrieve only one subpool at a time, starting with the smallest. See *IMS Version 9: Application Programming: Design Guide* for further information about the statistics returned by STAT.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L      STAT  DBASF
L      STAT  VBASS
L      STAT  VBASS
L      STAT  VBASS
L      STAT  VBASS
```

**SYNC Call:** Use a CALL FUNCTION statement to contain the SYNC function. The CALL DATA statement is optional.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L      SYNC
```

**Initial XRST Call:** Use a CALL FUNCTION statement to contain the XRST FUNCTION and a CALL DATA statement that contains a checkpoint ID of blanks to indicate that you are normally starting a program that uses symbolic checkpoints.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L      XRST                                     10101400
L      DATA
L      CKPT
L      DATA  YOURID01
```

**Basic XRST Call:** Use a CALL FUNCTION statement to contain the XRST function and a CALL DATA statement to contain the checkpoint ID.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L      XRST                                     10101400
L      DATA  TESTCKPT
```

**Symbolic XRST Call:** Use a CALL FUNCTION statement to contain the XRST function, a CALL DATA statement to contain the checkpoint ID data, and one or more CALL DATA statements where the data is to be returned.

The XRST call is used with the symbolic CHKP call.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L      XRST
L      DATA  TSTCHKP2                               X
L      8 DATA  OVERLAY2                               X
L      16 DATA  OVERLAY2OVERLAY2
U EIGHT BYTES OF DATA (OVERLAY2) SHOULD BE OVERLAID WITH CHECKPOINTED DATA
U SIXTEEN BYTES OF DATA (OVERLAY2OVERLAY2) IS OVERLAID ALSO
```

## CALL FUNCTION Statement with Column-Specific SSAs

In this format, the SSA has intervening blanks between fields. Columns 24, 34, and 37 must contain blanks. Command codes are not permitted. Table 64 gives the format for the CALL FUNCTION statement with column-specific SSAs.

Table 64. CALL FUNCTION Statement (Column-Specific SSAs)

Column	Function	Code	Description
1	Identifies control statement	L	Call statement (see columns 10-13).

Table 64. CALL FUNCTION Statement (Column-Specific SSAs) (continued)

Column	Function	Code	Description
2	Reserved	<b>b</b>	
3	Reserved	<b>b</b>	
4	Reserved	<b>b</b>	
5-8	Repeat Count	<b>b</b>	If blank, repeat count defaults to 1.
		<b>nnnn</b>	'nnnn' is the number of times to repeat this call. Range 1 to 9999, right-justified but need not contain leading zeros.
10-13	Identifies DL/I call function	<b>b</b>	If blank, use function from previous CALL statement.
		<b>xxxx</b>	'xxxx' is a DL/I call function.
		<b>CONT</b>	Continuation indicator for SSAs too long for a single CALL FUNCTION statement. Column 72 of preceding CALL FUNCTION statement must contain a nonblank character. The next CALL statement should have CONT in columns 10 through 13 and the SSA should continue in column 16.
14-15	Reserved	<b>b</b>	
16-23	SSA name	<b>s-name</b>	Required if call contains SSA.
24	Reserved	<b>b</b>	Separator field.
25	Start character for SSA	<b>(</b>	Required if segment is qualified.
26-33	SSA field name	<b>f-name</b>	Required if segment is qualified.
34	Reserved	<b>b</b>	Separator field.
35-36	DL/I call operator(s)	<b>name</b>	Required if segment is qualified.
37	Reserved	<b>b</b>	Separator field.
38-nn	Field value	<b>nnnnn</b>	Required if segment is qualified. <b>Note:</b> Do not use '5D' or ')' in field value.
nn+1	End character for SSA	<b>)</b>	Required if segment is qualified.
72	Continuation column	<b>b</b>	No continuations for this statement.
		<b>x</b>	Alone, it indicates multiple SSAs each beginning in column 16 of successive statements. With CONT in columns 10-13 of the next statement, indicates a single SSA that is continued beginning in column 16 of the following statement
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

If a CALL FUNCTION statement contains multiple SSAs, the statement must have a nonblank character in column 72 and the next SSA must start in column 16 of the next statement. If a field value extends past column 71, put a nonblank character in column 72. In the next statement insert the keyword CONT in columns 10 through 13 and continue the field value starting at column 16. Maximum length for field value is 256 bytes, maximum size for an SSA is 290 bytes, and the maximum number of SSAs for this program is 15, which is the same as the IMS limit.

## DFSDDLTO Call Functions

The DFSDDLTO call functions were created for DFSDDLTO. They do not represent "valid" IMS calls and are not punched as output if DFSDDLTO encounters them while a CTL (PUNCH) statement is active. Table 65 on page 331 shows the special call functions of the CALL FUNCTION statement. Descriptions and examples of these special functions follow.

Table 65. CALL FUNCTION Statement with DFSDDLTO Call Functions

Column	Function	Code	Description
1	Identifies control statement	<b>L</b>	Call statement.
2-4	Reserved	<b>b</b>	
5-8	Repeat count	<b>b</b>	If blank, repeat count defaults to 1.
		<b>nnnn</b>	'nnnn' is the number of times to repeat this call. Range is 1 to 9999, right-justified but need not contain leading zeros.
9	Reserved	<b>b</b>	
10-15	Special call function	<b>STAKb</b>	Stack control statements for later execution.
		<b>ENDbb</b>	Stop stacking and begin execution.
		<b>SKIPb</b>	Skip statements until START function is encountered.
		<b>START</b>	Start processing statements again.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

**STAK/END (stacking) Control Statements**

With the STAK statement, you repeat a series of statements that were read from SYSIN and held in memory. All control statements between the STAK statement and the END statement are read and saved. When DFSDDLTO encounters the END statement, it executes the series of calls as many times as specified in columns 5 through 8 of the STAK statement. STAK calls imbedded within another STAK cause the outer STAK call to be abnormally terminated.

**SKIP/START (skipping) Control Statements**

With the SKIP and START statements, you identify groups of statements that you do not want DFSDDLTO to process. These functions are normally read from SYSIN2 and provide a temporary override to an established SYSIN input stream. DFSDDLTO reads all control statements occurring between the SKIP and START statements, but takes no action. When DFSDDLTO encounters the START statement, it reads and processes the next statement normally.

**Examples of DFSDDLTO Call Functions**

**STAK/END Call:** The following example shows the STAK and END call functions.

```
//BATCH.SYSIN DD *
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
O SNAP= ,ABORT=0
S 1 1 1 1 1
L GU SEGA (KEYA =A300)
L 0003 STAK
WTO THIS IS PART OF THE STAK
T THIS COMMENT IS PART OF THE STAK
L GN
L END
U THIS COMMENT SHOULD GET PRINTED AFTER THE STAK IS DONE 3 TIMES
L 0020 GN
/*
```

**SKIP/START Call:** The following example demonstrates the use of the SKIP and START call functions in SYSIN2 to override and stop the processing of the STAK

and END call functions in SYSIN. DFSDDLTO executes the GU call function in SYSIN, skips the processing of STACK, WTO, T comment, GN, and END in SYSIN, and goes to the COMMENT.

```
//BATCH.SYSIN DD *
|-----1-----2-----3-----4-----5-----6-----7-----<
O SNAP= ,ABORT=0
S 1 1 1 1 1
L GU SEGA (KEYA =A300)
L 0003 STAK
WTO THIS IS PART OF THE STAK
T THIS COMMENT IS PART OF THE STAK
L GN
L END
U THIS COMMENT SHOULD GET PRINTED AFTER THE STAK IS DONE 3 TIMES
L 0020 GN
/*
//BATCH.SYSIN2 DD *
|-----1-----2-----3-----4-----5-----6-----7-----<
L SKIP
L START
U THIS COMMENT SHOULD REPLACE THE STAK COMMENT
U *****THIS COMMENT SHOULD GET PRINTED BECAUSE OF SYSIN2*****
/*
```

---

## COMMENT Statement

Use the COMMENT statement to print comments in the output data. The two types of COMMENT statements, conditional and unconditional, are described below. Table 66 shows the format of the COMMENT statement.

### Conditional COMMENT Statement

You can use up to five conditional COMMENT statements per call; no continuation mark is required in column 72. Code the statements in the DFSDDLTO stream before the call they are to document. Conditional COMMENTS are read and held until a CALL is read and executed. (If a COMPARE statement follows the CALL, conditional COMMENTS are held until after the comparison is completed.) You control whether the conditional comments are printed with column 3 of the STATUS statement. DFSDDLTO prints the statements according to the STATUS statement in the following order: conditional COMMENTS, the CALL, and the COMPARE(s). The time and date are also printed with each conditional COMMENT statement.

### Unconditional COMMENT Statement

You can use any number of unconditional COMMENT statements. Code them in the DFSDDLTO stream before the call they are to document. The time and date are printed with each unconditional COMMENT statement. Table 66 lists the column number, function, code, and description

Table 66. COMMENT Statement

Column	Function	Code	Description
1	Identifies control statement	<b>T</b>	Conditional comment statement.
		<b>U</b>	Unconditional comment statement.
2-72	Comment data		Any relevant comment.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.



## Example of COMMENT Statement

**T/U Comment Calls:** The following example shows the T and U comment calls.

```
//BATCH.SYSIN DD *
|-----1-----2-----3-----4-----5-----6-----7-----<
O SNAP= ,ABORT=0
S 1 1 1 1 1
L GU SEGB (KEYA =A400)
T THIS COMMENT IS A CONDITIONAL COMMENT FOR THE FIRST GN
L GN
U THIS COMMENT IS AN UNCONDITIONAL COMMENT FOR THE SECOND GN
L 0020 GN
/*
```

---

## COMPARE Statement

The COMPARE statement compares the actual results of a call with the expected results. The three types of COMPARE statements are the COMPARE PCB, COMPARE DATA, and COMPARE AIB.

When you use the COMPARE PCB, COMPARE DATA, and COMPARE AIB statements you must:

- Code COMPARE statements in the DFSDDLTO stream immediately after either the last continuation, if any, of the CALL DATA statement or another COMPARE statement.
- Specify the print option for the COMPARE statements in column 7 of the STATUS statement.

For all three COMPARE statements:

- The condition code returned for a COMPARE gives the total number of unequal comparisons.
- For single fixed-length segments, DFSDDLTO uses the comparison length to perform comparisons if you provide a length. The length comparison option (column 3) is not applicable.

When you use the COMPARE PCB statement and you want a snap dump when there is an unequal comparison, request it on the COMPARE PCB statement. A snap dump to a log with SNAP ID COMPxxxx is issued along with the snap dump options specified in column 3 of the COMPARE PCB statement.

The numeric part of the SNAP ID is initially set to 0000 and is incremented by 1 for each SNAP resulting from an unequal comparison.

## COMPARE DATA Statement

The COMPARE DATA statement is optional. It compares the segment returned by IMS to the data in the statement to verify that the correct segment was retrieved. Table 67 gives the format of the COMPARE DATA statement.

Table 67. COMPARE DATA Statement

Column	Function	Code	Description
1	Identifies control statement	<b>E</b>	COMPARE statement.
2	Reserved	<b>b</b>	

Table 67. COMPARE DATA Statement (continued)

Column	Function	Code	Description
3	Length comparison option	<b>b</b>	For fixed-length segments or if the LL field of the segment is not included in the comparison; only the data is compared.
		<b>L</b>	The length in columns 5-8 is converted to binary and compared against the LL field of the segment.
4	Segment length option	<b>b</b>	
		<b>V</b>	For a variable-length segment only, or for the first variable-length segment of multiple variable-length segments in a path call, or for a concatenated logical child/logical parent segment.
		<b>M</b>	For the second or subsequent variable-length segment of a path call, or for a concatenated logical child/logical parent segment.
		<b>P</b>	For fixed-length segments in path calls.
5-8	Comparison length	<b>Z</b>	For message segment.
		<b>nnnn</b>	Length to be used for comparison. (Required for length options V, M, and P if L is coded in column 3.)
9	Reserved	<b>b</b>	
10-13	Identifies type of statement	<b>DATA</b>	Required for the first I/O COMPARE statement and the first statement of a new segment if data from previous I/O COMPARE statement is not continued.
14-15	Reserved	<b>b</b>	
16-71	String of data		Data against which the segment in the I/O area is to be compared.
72	Continuation column	<b>b</b>	If blank, data is NOT continued.
		<b>x</b>	If not blank, data will be continued, starting in columns 16-71 of the subsequent statements for a maximum of 3840 bytes.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

**Notes:**

- If you code an L in column 3, the value in columns 5 through 8 is converted to binary and compared against the LL field of the returned segment. If you leave column 3 blank and the segment is not in a path call, then the value in columns 5 through 8 is used as the length of the comparison.
- If you code column 4 with a V, P, or M, you must enter a value in columns 5 through 8.
- If this is a path call comparison, code a P in column 4. The value in columns 5 through 8 must be the exact length of the fixed segment used in the path call.
- If you specify the length of the segment, this length is used in the COMPARE and in the display. If you do not specify a length, DFSDDL0 uses the shorter of the following for the length of the comparison and display:
  - The length of data supplied in the I/O area by IMS
  - The number of DATA statements read times 56

## COMPARE AIB Statement

The COMPARE AIB statement is optional. You can use it to compare values returned to the AIB by IMS. Table 68 shows the format of the COMPARE AIB statement.

Table 68. COMPARE AIB Statement

Column	Function	Code	Description
1	Identifies control statement	<b>E</b>	COMPARE statement.
2	Hold compare option	<b>H</b>	Hold COMPARE statement; see the paragraph below for details.
		<b>b</b>	Reset hold condition for a single COMPARE statement.
3	Reserved	<b>b</b>	
4-6	AIB compare	<b>AIB</b>	Identifies an AIB compare.
7	Reserved	<b>b</b>	
8-11	Return code	<b>xxxx</b>	Allow specified return code only.
12	Reserved		
13-16	Reason code	<b>xxxx</b>	Allow specified reason code only.
17-72	Reserved	<b>b</b>	<b>b</b>
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

To execute the same COMPARE AIB after a series of calls, put an H in column 2. When you specify an H, the COMPARE statement executes after each call. The H COMPARE statement is particularly useful when comparing with the same status code on repeated calls. The H COMPARE statement stays in effect until another COMPARE AIB statement is read.

## COMPARE PCB Statement

The COMPARE PCB statement is optional. You can use it to compare values returned to the PCB by IMS or to print blocks or buffer pool. Table 69 shows the format of the COMPARE PCB statement.

Table 69. COMPARE PCB Statement

Column	Function	Code	Description
1	Identifies control statement	<b>E</b>	COMPARE statement.
2	Hold compare option	<b>H</b>	Hold compare statement.
		<b>b</b>	Reset hold condition for a single COMPARE statement.
3	Snap dump options (if compare was unequal)	<b>b</b>	Use default value. (You can change the default value or turn off the option by coding the value in an OPTION statement.)
		<b>1</b>	The complete I/O buffer pool.
		<b>2</b>	The entire region (batch regions only).
		<b>4</b>	The DL/I blocks.
		<b>8</b>	Terminate the job step on miscompare of DATA or PCB.

Table 69. COMPARE PCB Statement (continued)

Column	Function	Code	Description
		<b>S</b>	To SNAP subpools 0 through 127. Requests for multiple SNAP dump options can be obtained by summing their respective hexadecimal values. If anything other than a blank, 1-9, A-F, or S is coded in column 3, the SNAP dump option is ignored.
4	Extended SNAP <sup>1</sup> options	<b>b</b>	Ignore extended option.
		<b>P</b>	SNAP the complete buffer pool (batch).
		<b>S</b>	SNAP subpools 0 through 127 (batch).  An area is never snapped twice. The SNAP option is a combination of columns 3 (SNAP dump option) and 4 (extended SNAP option).
5-6	Segment level	<b>nn</b>	'nn' is the segment level for COMPARE PCB. A leading zero is required.
7	Reserved	<b>b</b>	
8-9	Status code	<b>b</b>	Allow blank status code only.
		<b>xx</b>	Allow specified status code only.
		<b>XX</b>	Do not check status code.
		<b>OK</b>	Allow the following: blank, GA, GC, or GK.
10	Reserved	<b>b</b>	
11-18	Segment name User Identification	<b>xxxxxxxx</b>	Segment name for DB PCB compare.  Logical terminal for I/O.  Destination for ALT PCB.
19	Reserved	<b>b</b>	
20-23	Length of key	<b>nnnn</b>	'nnnn' is length of the feedback key.
24-71 or	Concatenated key		Concatenated key feedback for DB PCB compare.
24-31	User ID		User identification for TP PCB.
72	Continuation column	<b>b</b>	If blank, key feedback is <i>not</i> continued.
		<b>x</b>	If not blank, key feedback is continued, starting in columns 16-71 of subsequent statements.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

**Note:**

1. SNAP is a Product-sensitive programming interface.

Blank fields are not compared to the corresponding field in the PCB, except for the status code field. (Blanks represent a valid status code.) To accept the status codes blank, GA, GC, or GK as a group, put OK in columns 8 and 9. To stop comparisons of status codes, put XX in columns 8 and 9.

To execute the same compare after a series of calls, put an H in column 2. This executes the COMPARE statement after each call. This is particularly useful to compare to a blank status code only when loading a database. The H COMPARE statement stays in effect until another COMPARE PCB statement is read.

## Examples of COMPARE DATA and PCB Statements

**COMPARE PCB Statement for Blank Status Code:** The COMPARE PCB statement is coded blank. It checks a blank status code for the GU.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          GU                                     10101100
E                                               10101200
```

**COMPARE PCB Statement for SSA Level, Status Code, Segment Name, Concatenated Key Length, and Concatenated Key:** The COMPARE PCB statement is a request to compare the SSA level, a status code of OK (which includes blank, GA, GC, and GK), segment name of SEGA, concatenated key length of 0004, and a concatenated key of A100.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          GU
E  01 OK SEGA      0004A100
```

**COMPARE PCB Statement for SSA Level, Status Code, Segment Name, Concatenated Key Length, and Concatenated Key:** The COMPARE PCB statement causes the job step to terminate based on the 8 in column 3 when any of the fields in the COMPARE PCB statement are not equal to the corresponding field in the PCB.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          GU                                     10105100
E  8 01 OK SEGK      0004A100                    10105200
```

**COMPARE PCB Statement for Status Code with Hold Compare:** The COMPARE PCB statement is a request to compare the status code of OK (which includes blank, GA, GC, and GK) and hold that compare until the next COMPARE PCB statement. The compare of OK is used on GN following GU and is also used on a GN that has a request to be repeated six times.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          GU   SEGA   (KEYA   = A300)          20201100
L          GN                                     20201300
EH         OK                                     20201400
L   0006 GN                                     20201500
```

**COMPARE DATA Statement for Fixed-Length Segment:** The COMPARE DATA statement is a request to compare the data returned. 72 bytes of data are compared.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          GU
E          DATA A100A100A100A100A100A100A100A100A100A100A100A100A100A100X10102200
E          A100A100A100A100                                     10102300
```

**COMPARE DATA Statement for Fixed-Length Data for 64 Bytes:** The COMPARE DATA statement is a request to compare 64 bytes of the data against the data returned.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
L          GU                                     10101600
E  0064 DATA A100A100A100A100A100A100A100A100A100A100A100A100A100A100X10101700
E          A100A100B111B111                             10101800
```

**COMPARE DATA Statement for Fixed-Length Data for 72 Bytes:** The COMPARE DATA statement is a request to compare 72 bytes of the data against the data returned.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L          GU                                     10103900
E LP0072 DATA A100A100A100A100A100A100A100A100A100A100A100A100A100A100X10104000
E          A100A100A100A100                                     10104100
```

**COMPARE DATA Statement for Variable-Length Data of Multiple-Segments Data and Length Fields:** The COMPARE DATA statement is a request to compare 36 bytes of the data against the data returned for segment 1 and 16 bytes of data for segment 2. It compares the length fields of both segments.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L          ISRT D          (DSS      = DSS01)          X38005500
L          DJ          (DJSS     = DJSS01)          X38005600
L          QAJAXQAJ          38005700
L V0036 DATA QSS02QASS02QAJSS01QAJASS97*IQAJA**      *38005800
L M0016 DATA QAJSS01*IQAJ**          38005850
L          GHU D          (DSS      = DSS01)          X38006000
L          DJ          (DJSS     = DJSS01)          X38006100
L          QAJAXQAJ (QAJASS = QAJASS97)          38006200
E LV0036 DATA QSS02QASS02QAJSS01QAJASS97*IQAJA**      *38006300
E LM0016 DATA QAJSS01*2QAJ**          38006350
```

**COMPARE DATA Statement for Variable-Length Data of Multiple Segments with no Length Field COMPARE:** The COMPARE DATA statement is a request to compare 36 bytes of the data against the data returned for segment 1 and 16 bytes of data for segment 2 with no length field compares of either segment.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L          ISRT D          (DSS      = DSS01)          X38005500
L          DJ          (DJSS     = DJSS01)          X38005600
L          QAJAXQAJ          38005700
L V0036 DATA QSS02QASS02QAJSS01QAJASS97*IQAJA**      *38005800
L M0016 DATA QAJSS01*IQAJ**          38005850
L          GHU D          (DSS      = DSS01)          X38006000
L          DJ          (DJSS     = DJSS01)          X38006100
L          QAJAXQAJ (QAJASS = QAJASS97)          38006200
E V0036 DATA QSS02QASS02QAJSS01QAJASS97*IQAJA**      *38006300
M0016 DATA QAJSS01*2QAJ**          38006350
```

**COMPARE DATA Statement for Variable-Length Data of Multiple Segments and One Length Field COMPARE:** The COMPARE DATA statement is a request to compare 36 bytes of the data against the data returned for segment 1 and 16 bytes of data for segment 2. It compares the length field of segment 1 only.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
L          ISRT D          (DSS      = DSS01)          X38005500
L          DJ          (DJSS     = DJSS01)          X38005600
L          QAJAXQAJ          38005700
L V0036 DATA QSS02QASS02QAJSS01QAJASS97*IQAJA**      *38005800
L M0016 DATA QAJSS01*IQAJ**          38005850
L          GHU D          (DSS      = DSS01)          X38006000
L          DJ          (DJSS     = DJSS01)          X38006100
L          QAJAXQAJ (QAJASS = QAJASS97)          38006200
E LV0036 DATA QSS02QASS02QAJSS01QAJASS97*IQAJA**      *38006300
M0016 DATA QAJSS01*2QAJ**          38006350
```

## IGNORE Statement

DFSDDLTO ignores any statement with an N or a period (.) in column 1. You can use the N or . (period) to comment out a statement in either the SYSIN or SYSIN2 input streams. Using an N or . (period) in a SYSIN2 input stream causes the SYSIN input stream to be ignored as well. See “SYSIN2 DD Statement” on page 348 for information on how to override SYSIN input. Table 70 gives the format of the IGNORE statement. An example of the statement follows.

Table 70. IGNORE Statement

Column	Function	Code	Description
1	Identifies control statement	N or .	IGNORE statement.
2-72	Ignored		
73-80	Sequence indication	nnnnnnnn	For SYSIN2 statement override.

### Example of IGNORE (N or .)

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
. NOTHING IN THIS AREA WILL BE PROCESSED. ONLY THE SEQUENCE NUMBER      67101010
N WILL BE USED IF READ FROM SYSIN2 OR SYSIN.                               67101020
```

## OPTION Statement

Use the OPTION statement to override various default options. Use multiple OPTION statements if you cannot fit all the options you want in one statement. No continuation character is necessary. Once you set an option, it remains in effect until you specify another OPTION statement to change the first parameter. Table 71 shows the format of the OPTION statement. An example follows.

Table 71. OPTION Statement

Column	Function	Code	Description
1	Identifies control statement	O	OPTION statement (free-form parameter fields).
2	Reserved	b	b
3-72	Keyword parameters:		
	ABORT=	<ul style="list-style-type: none"> <li>• 0</li> <li>• 1 to 9999</li> </ul>	<ul style="list-style-type: none"> <li>• Turns the ABORT parameter off.</li> <li>• Number of unequal compares before aborting job. Initial default is 5.</li> </ul>
	LINECNT=	10 to 99	Number of lines printed per page. Must be filled with zeros. Initial default 54.
	SNAP <sup>1</sup>	x	SNAP option default, when results of compare are unequal. To turn the SNAP option off, code 'SNAP='. See “COMPARE PCB Statement” on page 335 for the appropriate values for this parameter. (Initial default is 5 if this option is not coded. This causes the I/O buffer pool and the DL/I blocks to be dumped with a SNAP call.)
	DUMP/NODUMP		Produce/do not produce dump if job abends. Default is NODUMP.

Table 71. OPTION Statement (continued)

Column	Function	Code	Description
	LCASE=	<ul style="list-style-type: none"> <li>• H</li> <li>• C</li> </ul>	<ul style="list-style-type: none"> <li>• Hexadecimal representation for lower case characters. This is the initial default.</li> <li>• Character representation for lower case characters.</li> </ul>
	STATCD/NOSTATCD		Issue/do not issue an error message for the internal, end-of-job stat call that does not receive a blank or GA status code. NOSTATCD is the default.
	ABU249/NOABU249		Issue/do not issue a DFSDDLTO ABENDU0249 when an invalid status code is returned for any of the internal end-of-job stat calls in a batch environment. NOABU249 is the default.
73 - 80	Sequence indication	nnnnnnnn	For SYSIN2 statement override.

**Note:**

1. SNAP is a Product-sensitive programming interface.

OPTION statement parameters can be separated by commas.

### Example of OPTION Control Statement

```
|-----1-----+-----2-----+-----3-----+-----4-----+-----5-----+-----6-----+-----7-----+-----<
0 ABORT=5,DUMP,LINECNT=54,SPA=4096,SNAP=5                                67101010
```

### PUNCH Statement

The PUNCH CTL statement allows you to produce an output data set consisting of COMPARE PCB statements, COMPARE DATA statements, COMPARE AIB statements, other control statements (with the exceptions noted below), or combinations of the above. Table 72 shows the format and keyword parameters for the PUNCH CTL statement.

Table 72. PUNCH CTL Statement

Column	Function	Code	Description
1-3	Identifies control statement	<b>CTL</b>	PUNCH statement.
4-9	Reserved	<b>b</b>	
10-13	Punch control	<b>PUNC</b>	Begin punching (no default values).
		<b>NPUN</b>	Stop punching (default value).
14-15	Reserved	<b>b</b>	
16-72	Keyword parameters:		



Table 72. PUNCH CTL Statement (continued)

Column	Function	Code	Description
	OTHER		Reproduces all input control statements except: <ul style="list-style-type: none"> <li>• CTL (PUNCH) statements.</li> <li>• N or . (IGNORE) statements.</li> <li>• COMPARE statements.</li> <li>• CALL statements with functions of SKIP and START. Any control statements that appear between SKIP and START CALLs are not punched. (See "SKIP/START (skipping) Control Statements" on page 331).</li> <li>• CALL statements with functions of STAK and END. Control statements that appear between STAK and END CALLs are saved and then punched the number of times indicated in the STAK CALL. (See "STAK/END (stacking) Control Statements" on page 331).</li> </ul>
	DATAL		Create a full data COMPARE using all of the data returned to the I/O area. Multiple COMPARE statements and continuations are produced as needed.
	DATAS		Create a single data COMPARE statement using only the first 56 bytes of data returned to the I/O area.
	PCBL		Create a full PCB COMPARE using the complete key feedback area returned in the PCB. Multiple COMPARE statements and continuations are produced as needed.
	PCBS		Create a single PCB COMPARE statement using only the first 48 bytes of the key feedback area returned in the PCB.
	SYNC/NOSYNC		If a GB status code is returned on a Fast Path call while in STAK, but prior to exiting STAK, this function issues or does not issue SYNC.
	START=		00000001 to 99999999.  This is the starting sequence number to be used for the punched statements. Eight numeric bytes must be coded.
	INCR=		1 to 9999.  Increment the sequence number of each punched statement by this value. Leading zeros are not required.
	AIB		Create an AIB COMPARE statement.
73-80	Sequence indication	nnnnnnnn	For SYSIN2 statement override.

To change the punch control options while processing a single DFSDDLTO input stream, always use PUNCH CTL statements in pairs of PUNC and NPUN.

One way to use the PUNCH CTL statement is as follows:

1. Code only the CALL statements for a new test. Do not code the COMPARE statements.
2. Verify that each call was executed correctly.
3. Make another run using the PUNCH CTL statement to have DFSDDLTO merge the proper COMPARE statements and produce a new output data set that can be used as input for subsequent regression tests.

You can also use PUNCH CTL if segments in an existing database are changed. The control statement causes DFSDDLTO to produce a new test data set that has the correct COMPARE statements rather than you having to manually change the COMPARE statements.

Parameters in the CTL statement must be the same length as described in Table 72, and they must be separated by commas.

### Example of PUNCH CTL Statement

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
CTL      PUNC  PCBS,DATAS,OTHER,START=00000010,INCR=0010      33212010
CTL      NPUN                                     33212020
```

The DD statement for the output data set is labeled PUNCHDD. The data sets are fixed block with LRECL=80. Block size as specified on the DD statement is used. If not specified, the block size is set to 80. If the program is unable to open PUNCHDD, DFSDDLTO issues abend 251.

### Example of PUNCH CTL Statement for All Parameters

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
CTL      PUNC  OTHER,DATAL,PCBL,START=00000001,INCR=1000,AIB      33212010
```

## STATUS Statement

With the STATUS statement, you establish print options and name the PCB that you want subsequent calls to be issued against. Table 73 shows the format of the STATUS statement.

Table 73. STATUS Statement

Column	Function	Code	Description
1	Identifies control statement	<b>S</b>	STATUS statement.
2	Output device option	<b>b</b>	Use PRINTDD when in a DL/I region; use I/O PCB in MPP region.
		<b>1</b>	Use PRINTDD in MPP region if DD statement is provided; otherwise, use I/O PCB.
		<b>A</b>	Same as if 1, and disregard all other fields in this STATUS statement.
3	Print comment option	<b>b</b>	Do not print.
		<b>1</b>	Print for each call.
		<b>2</b>	Print only if compare done and unequal.
4	Print AIB option	<b>b</b>	Do not print.
		<b>1</b>	Print for each call.

Table 73. STATUS Statement (continued)

Column	Function	Code	Description
		<b>2</b>	Print only if compare done and unequal.
5	Print call option	<b>b</b>	Do not print.
		<b>1</b>	Print for each call.
		<b>2</b>	Print only if compare done and unequal.
6	Reserved	<b>b</b>	
7	Print compare option	<b>b</b>	Do not print.
		<b>1</b>	Print for each call.
		<b>2</b>	Print only if compare done and unequal.
8	Reserved	<b>b</b>	
9	Print PCB option	<b>b</b>	Do not print.
		<b>1</b>	Print for each call.
		<b>2</b>	Print only if compare done and unequal.
10	Reserved	<b>b</b>	
11	Print segment option	<b>b</b>	Do not print.
		<b>1</b>	Print for each call.
		<b>2</b>	Print only if compare done and unequal.
12	Set task and real time	<b>b</b>	Do not time
		<b>1</b>	Time each call.
		<b>2</b>	Time each call if compare done and unequal.
13-14	Reserved	<b>b</b>	
15	PCB selection option	<b>1</b>	PCB name passed in columns 16-23 (use option 1).
		<b>2</b>	DBD name passed in columns 16-23 (use option 2).
		<b>3</b>	Relative DB PCB passed in columns 16-23 (use option 3).
		<b>4</b>	Relative PCB passed in columns 16-23 (use option 4).
		<b>5</b>	\$LISTALL passed in columns 16-23 (use option 5).
		<b>b</b>	If column 15 is blank, DFSDDLTO selects options 2 through 5 based on content of columns 16-23.
Opt. 1 16-23	PCB selection PCB name	<b>alpha</b>	These columns must contain the name of the label on the PCB at PSBGEN, or the name specified on the PCBNAME= operand for the PCB at PSBGEN time.
Opt. 2 16-23	PCB selection DBD name	<b>b</b> <b>alpha</b>	The default PCB is the first database PCB in the PSB. If columns 16-23 are blank, current PCB is used. If DBD name is specified, this must be the name of a database DBD in the PSB.
Opt. 3 16-18 19-23	PCB selection Relative position of PCB in PSB	<b>b</b> <b>numeric</b>	When columns 16 through 18 are blank, columns (19-23) of this field are interpreted as the relative number of the DB PCB in the PSB. This number must be right-justified to column 23, but need not contain leading zeros.

Table 73. STATUS Statement (continued)

Column	Function	Code	Description
Opt. 4 16-18 19-23	PCB selection I/O PCB Relative position of PCB in PSB	<b>TPb</b> <b>numeric</b>	When columns 16 through 18 = 'TPb', columns (19-23) of this field are interpreted as the relative number of the PCB from the start of the PCB list. This number must be right-justified to column 23, but need not contain leading zeros. I/O PCB is always the first PCB in the PCB list in this program.
Opt. 5 16-23	List all PCBs in the PSB	<b>\$LISTALL</b>	Prints out all PCBs in the PSB for test script.
24	Print status option	<b>b</b>	Use print options to print this STATUS statement.
		<b>1</b>	Do not use print options in this statement; print this STATUS statement.
		<b>2</b>	Do not print this STATUS statement but use print options in this statement.
		<b>3</b>	Do not print this STATUS statement and do not use print options in this statement.
25-28	PCB processing option	<b>xxxx</b>	This is optional and is only used when two PCBs have the same name but different processing options. If not blank, it is used in addition to the PCB name in columns 16 through 23 to select which PCB in the PSB to use.
29	Reserved	<b>b</b>	
30-32	AIB interface	<b>AIB</b>	Indicates that the AIB interface is used and the AIB is passed rather than passing the PCB. (Passing the PCB is the default.) <b>Note:</b> When the AIB interface is used, the PCB must be defined at PSBGEN with PCBNAME=name. IOPCB is the PCB name used for all I/O PCBs. DFSDDLTO recognizes that name when column 15 contains a 1 and columns 16 through 23 contain IOPCB.
33	Reserved		
37-72	Reserved		
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

If DFSDDLTO does not encounter a STATUS statement, all default print options (columns 3 through 12) are 2 and the default output device option (column 2) is 1. You can code a STATUS statement before any call sequence in the input stream, changing either the PCB to be referenced or the print options.

The referenced PCB stays in effect until a subsequent STATUS statement selects another PCB. However, a call that must be issued against an I/O PCB (such as LOG) uses the I/O PCB for that call. After the call, the PCB changes back to the original PCB.

## Examples of STATUS Statement

**To Print Each CALL Statement:** The following STATUS statement tells DFSDDLTO to print these options: COMMENTS, CALL, COMPARE, PCB, and SEGMENT DATA for all calls.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
S 1 1 1 1 1
```

**To Print Each CALL Statement, Select a PCB:** The following STATUS statements tell DFSDDLT0 to print the COMMENTS, CALL, COMPARE, PCB, and SEGMENT DATA options for all calls, and select a PCB.

The 1 in column 15 is required for PCBNAME. If omitted, the PCBNAME is treated as a DBDNAME.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
S 1 1 1 1 1 1PCBNAME
```

```
|-----1-----2-----3-----4-----5-----6-----7-----<
S 1 1 1 1 1 1PCBNAME AIBb
```

**To print each CALL statement, select PCB based on a DBD name:** The following STATUS statements tell DFSDDLT0 to print the COMMENTS, CALL, COMPARE, PCB, and SEGMENT DATA options for all calls, and select a PCB by a DBD name.

The 2 in column 15 is optional.

```
|-----1-----2-----3-----4-----5-----6-----7-----<
S 1 1 1 1 1 2DBDNAME
```

```
|-----1-----2-----3-----4-----5-----6-----7-----<
S 1 1 1 1 1 2DBDNAME AIBb
```

If you do not use the AIB interface, you do not need to change STATUS statement input to existing streams; existing call functions will work just as they have in the past. However, if you want to use the AIB interface, you must change the STATUS statement input to existing streams to include AIB in columns 30 through 32. The existing DBD name, Relative DB PCB, and Relative PCB will work if columns 30 through 32 contain AIB and the PCB has been defined at PSBGEN with PCBNAME=name.

---

## WTO Statement

The WTO (Write to Operator) statement sends a message to the z/OS console without waiting for a reply. Table 74 shows the format for the WTO statement.

Table 74. WTO Statement

Column	Function	Code	Description
1-3	Identifies control statement	<b>WTO</b>	WTO statement.
4	Reserved	<b>b</b>	
5-72	Message to send		Message to be written to the system console.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

## Example of WTO Statement

This WTO statement, in this example, sends a message to the z/OS console and continues the test stream.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
WTO AT A "WTO" WITHIN TEST STREAM --WTO NUMBER 1-- TEST STARTED
```

---

## WTOR Statement

The WTOR (Write to Operator with Reply) statement sends a message to the z/OS system console and waits for a reply. Table 75 shows the format of the WTOR statement.

Table 75. WTOR Statement

Column	Function	Code	Description
1-4	Identifies control statement	<b>WTOR</b>	WTOR statement.
5	Reserved	<b>b</b>	
6-72	Message to send		Message to be written to the system console.
73-80	Sequence indication	<b>nnnnnnnn</b>	For SYSIN2 statement override.

## Example of WTOR Statement

This WTOR statement causes the test stream to hold until DFSDDLTO receives a response from the z/OS console operator. Any response is valid.

```
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
WTOR AT A "WTOR" WITHIN TEST STREAM - ANY RESPONSE WILL CONTINUE
```

---

## JCL Requirements

This section defines the DD statements that DFSDDLTO uses. Execution JCL depends on the installation data set naming standards as well as the IMS environment (batch or online). See Figure 73 on page 347.

```

//SAMPLE JOB ACCOUNTING,NAME,MSGLEVEL=(1,1),MSGCLASS=3,PRTY=8          33001100
//GET EXEC PGM=DFSRRCO0,PARM='DLI,DFSDDLTO,PSBNAME'                    33001200
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR                                33001300
//IMS DD DSN=IMS2.PSBLIB,DISP=(SHR,PASS)                             33001400
// DD DSN=IMS2.DBDLIB,DISP=(SHR,PASS)                                33001500
//DDCARD DD DSN=DATASET,DISP=(OLD,KEEP)                              33001600
//IEFRDER DD DUMMY                                                    33001700
//PRINTDD DD SYSOUT=A                                                  33001800
//SYSUDUMP DD SYSOUT=A                                                33001900
//SYSIN DD *                                                            33002000
|-----1-----2-----3-----4-----5-----6-----7-----<
U THIS IS PART OF AN EXAMPLE                                          33002100
S 1 1 1 1 1 PCB-NAME                                                33002200
L GU                                                                    33002300
/*
//SYSIN2 DD *
|-----1-----2-----3-----4-----5-----6-----7-----<
ABEND                                                                    33002300
/*

```

Figure 73. Example JCL Code for DD Statement Definition

Figure 74 is an example of coding JCL for DFSDDLTO in a BMP. Use of a procedure is optional and is only shown here as an example.

```

//SAMPLE JOB ACCOUNTING,NAME,MSGLEVEL=(1,1),MSGCLASS=A                00010047
//*****
/* BATCH DL/I JOB TO RUN FOR RSR TESTING *
//*****
//BMP EXEC IMSBATCH,MBR=DFSDDLTO,PSB=PSBNAME
//BMP.PRINTDD DD SYSOUT=A
//BMP.PUNCHDD DD SYSOUT=B
//BMP.SYSIN DD *
U ***THIS IS PART OF AN EXAMPLE OF SYSIN DATA                        00010000
S 1 1 1 1 1 1                                                         00030000
L GU                                                                    00040000
L 0099 GN                                                                00050000
/*
|-----1-----2-----3-----4-----5-----6-----7-----<
//BMP.SYSIN2 DD *
U ***THIS IS PART OF AN EXAMPLE OF SYSIN2 DATA *****              00020000
ABEND                                                                    00050000
/*

```

Figure 74. Example JCL Code for DFSDDLTO in a BMP

## SYSIN DD Statement

The data set specified by the SYSIN DD statement is the normal input data set for DFSDDLTO. When processing input data that is on direct-access or tape, you may want to override certain control statements in the SYSIN input stream or to add other control statements to it. You do this with a SYSIN2 DD statement and the control statement sequence numbers.

Sequence numbers in columns 73 to 80 for SYSIN data are optional unless a SYSIN2 override is used. If a SYSIN2 override is used, follow the directions for using sequence numbers as described in “SYSIN2 DD Statement” on page 348.

## SYSIN2 DD Statement

DFSDDLTO does not require the SYSIN2 DD statement, but if it is present in the JCL, DFSDDLTO will read and process the specified data sets. When using SYSIN2, the following items apply:

- The SYSIN DD data set is the primary input. DFSDDLTO attempts to insert the SYSIN2 control statements into the SYSIN DD data set.
- You must code the control groups and sequence numbers properly in columns 73 to 80 or the merging process will not work.
- Columns 73 and 74 indicate the control group of the statement.
- Columns 75 to 80 indicate the sequence number of the statement.
- Sequence numbers *must* be in numeric order within their control group.
- Control groups in SYSIN2 must match the SYSIN control groups, although SYSIN2 does not have to use all the control groups used in SYSIN. DFSDDLTO does not require that control groups be in numerical order, but the control groups in SYSIN2 must be in the same order as those in SYSIN.
- When DFSDDLTO matches a control group in SYSIN and SYSIN2, it processes the statements by sequence number. SYSIN2 statements falling before or after a SYSIN statement are merged accordingly.
- If the sequence number of a SYSIN2 statement matches the sequence number of a SYSIN statement in its control group, the SYSIN2 overrides the SYSIN.
- If the program reaches the end of SYSIN before it reaches the end of SYSIN2, it processes the records of SYSIN2 as if they were an extension of SYSIN.
- Replacement or merging occurs only during the current run. The original SYSIN data is not changed.
- During merge, if one of the control statements contains blanks in columns 73 through 80, DFSDDLTO discards the statement containing blanks, sends a message to PRINTDD, and continues the merge until end-of-file is reached.

## PRINTDD DD Statement

The PRINTDD DD statement defines the output data set for DFSDDLTO. The output data set might include displays of control blocks written to the data set as the result of SNAP calls. The data set defined by the PRINTDD DD statement must conform to the z/OS SNAP data set requirements.

## PUNCHDD DD Statement

The DD statement for the output data set is labeled PUNCHDD. The data sets are fixed block with LRECL=80. Block size as specified on the DD statement is used; if not specified, the block size is set to 80. If the program is unable to open PUNCHDD, DFSDDLTO issues abend 251. Here is an example of the PUNCHDD DD statement.

```
//PUNCHDD DD SYSOUT=B
```

## Using the PREINIT Parameter for DFSDDLTO Input Restart

You use the DFSDDLTO restart function to restart a DFSDDLTO input stream within the same dependent region that the input stream was running prior to the restart. The PREINIT parameter in the EXEC statement invokes the restart function. Code the PREINIT parameter of DFSMPR procedure as PREINIT=xx, where xx is the two-character suffix of the DFSINTxx PROCLIB member. (PREINIT=DL refers to the default IMS.PROCLIB member.)



The PREINIT process establishes a checkpoint field for each active IMS region. This field is updated with the sequence number of each GU call to an I/O PCB as it is processed. For this reason, sequence numbers are required for all such GU calls that are used. On a restart, if the checkpoint field contains a sequence number, the DFSDDLTO stream starts at the next GU call to an I/O PCB following the sequence number in the checkpoint field; otherwise the DFSDDLTO stream starts from the beginning.

The DFSDDLSI module and the default IMS.

member, DFSINTDL, are shipped with IMS and are installed as part of normal IMS installation.

The following code shows examples of SYSIN/SYSIN2 and PREINIT.

```
//TSTPGM   JOB CARD
//DDLTTST EXEC DFSMPR,PREINIT=DL
//MPP.SYSIN DD *
|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
S11 1 1 1 1   TP    1                                01000000
OPTIONS SNAP= ,ABORT=9999                            01000010
U*****                                              01000040
S11 1 1 1 1   TP    1                                01000050
L          GU                                         01000060
E          OK                                         01000070
S11 1 1 1 1   DBPCBXXX                               01000080
L          GU                                         01000090
E          DATA  A  INIT-LOAD UOW                   01000100
E  01  ROOTSEG1 0008A 0004D                          01000110
S11 1 1 1 1   TP    1                                01000120
L          ISRT                                       01000130
L  Z0080 DATA -SYNC INTERVAL 1  SEG 1 -MESSAGE 1    X01000140
L P      DATA 111111111111111111111111111111111111 01000150
L          ISRT                                       01000160
L  Z0080 DATA -SYNC INTERVAL 1  SEG 2 -END EOM 1    X01000170
L P      DATA 111111111111111111111111111111111111 01000180
U*****                                              01000190
U*  ENDING FIRST SYNC INTERVAL                       01000200
U*****                                              01000210
L          GU                                         01000220
E          QC                                          01000230
L          GU                                         01000240
E          OK                                         01000250
S11 1 1 1 1   DBPCBXXX                               01000260
WTO GETTING DATA BASE SEGMENT 1 FROM DBPCBXXX      01000270
L  U      GHU                                         01000280
E          DATA INIT-LOAD UOW. 1 A.P. 1             01000290
E          OK                                         01000300
L  U0003 GN                                           01000310
E          OK                                         01000320
S11 1 1 1 1   TP    1                                01000330
L          ISRT                                       01000340
L  Z0080 DATA -SYNC INTERVAL 2  SEG 1 -MESSAGE 1    X01000350
L P      DATA 222222222222222222222222222222222221 01000360
L          ISRT                                       01000370
L  Z0080 DATA -SYNC INTERVAL 2  SEG 2 -END EOM 1    X01000380
L P      DATA 222222222222222222222222222222222211 01000390
U*****                                              01000400
U*  ENDING SECOND SYNC INTERVAL                      01000410
U*****                                              01000420
L          GU                                         01000430
E          QC                                          01000440
L          GU                                         01000450
E          OK                                         01000460
S11 1 1 1 1   DBPCBXXX                               01000470
```



code is received, this is treated as the end of file. When input is from the I/O PCB, you can send output to PRINTDD by coding a 1 or an A in column 2 of the STATUS statement.

Because the input is in fixed form, it is difficult to key it from a terminal. To use DFSDDLT0 to test DL/I in a message region, execute another message program that reads control statements stored as a member of a partitioned set. Insert these control statements to an input transaction queue. IMS then schedules the program to process the transactions. This method allows you to use the same control statements to execute in any region type.

## Explanation of DFSDDLT0 Return Codes

A non-zero return code from DFSDDLT0 indicates the number of unequal comparisons that occurred during that time.

A return code of 0 (zero) from DFSDDLT0 does not necessarily mean that DFSDDLT0 executed without errors. There are several messages issued by DFSDDLT0 that do not change the return code, but do indicate some sort of error condition. This preserves the return code field for the unequal comparison count.

If an error message was issued during the run, a message ERRORS WERE DETECTED WITHIN THE INPUT STREAM. REVIEW OUTPUT TO DETERMINE ERRORS. appears at the end of the DFSDDLT0 output. You must examine the output to ensure DFSDDLT0 executed as expected.

## Hints on Using DFSDDLT0

This section describes loading a database, printing, retrieving, replacing, and deleting segments, regression testing, using a debugging aid, and verifying how a call is executed.

### To Load a Database

Use DFSDDLT0 for loading only very small databases because you must provide all the calls and data rather than have them generated. The following example shows CALL FUNCTION and CALL DATA statements that are used to load a database.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
0 SNAP= ,ABORT=0
S 1 2 2 1 1
L      ISRT COURSE
L      DATA FRENCH
L      ISRT COURSE
L      DATA COBOL
L      ISRT CLASS
L      DATA 12
L      ISRT CLASS
L      DATA 27
L      ISRT STUDENT
L      DATA SMITH          THERESE
L      ISRT STUDENT
L      DATA GRABOWSKY     MARION
    
```

### To Print the Segments in a Database

Use either of the following sequences of control statements to print the segments in a database.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
.* Use PRINTDD, print call, compare, and PCB if compare unequal
.* Do 1 Get Unique call
.* Hold PCB compare, End step if status code is not blank, GA, GC, GK
.* Do 9,999 Get Next calls
S  2 2 2 1  DBDNAME
L      GU
EH8   OK
L  9999 GN

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---<
.* Use PRINTDD, print call, compare, and PCB if compare unequal
.* Do 1 Get Unique call
.* Hold PCB compare, Halt GN calls when status code is GB.
.* Do 9,999 Get Next calls
S  2 2 2 1  DBDNAME
L      GU
EH     OK
L  9999 GN

```

Both of the above examples request the GN to be repeated 9999 times. Note that the first example uses a COMPARE PCB of EH8 while the second uses a COMPARE PCB of EH.

The difference between these two examples is that the first halts the job step the first time the status code is not blank, GA, GC, or GK. The second example halts repeating the GN and goes on to process any remaining DFSDDLTO control statements when a GB status code is returned or the GN has been repeated 9999 times.

## To Retrieve and Replace a Segment

Use the following sequence of control statements to retrieve and replace a segment.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---8---
S 1 1 1 1 1  COURSEDB
L      GHU  COURSE (TYPE   =FRENCH)           X
          CLASS (WEEK   =27)                 X
          STUDENT (NAME  =SMITH)
L      REPL
L      DATA SMITH      THERESE

```

## To Delete a Segment

Use the following sequence of control statements to delete a segment.

```

|---+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---8---
S 1 1 1 1 1  4
L      GHU  COURSE (TYPE   =FRENCH)           X
          CLASS *L                 X
          INSTRUC (NUMBER  =444)
L      DLET

```

## To Do Regression Testing

DFSDDLTO is ideal for doing regression testing. By using a known database, DFSDDLTO can issue calls and then compare the results of the call to expected results using COMPARE statements. The program then can determine if DL/I calls are executed correctly. If you code all the print options as 2's (print only if comparisons done and unequal), only the calls not properly satisfied are displayed.

## To Use as a Debugging Aid

When debugging a program, you usually need a print of the DL/I blocks. You can snap the blocks to a log data set at appropriate times by using a COMPARE statement that has an unequal compare in it. You can then print the blocks from the log. If you need the blocks even though the call executed correctly, such as for the call before the failing call, insert a SNAP function in the CALL statement in the input stream.

## To Verify How a Call Is Executed

Because it is very easy to execute a particular call, you can use DFSDDLTO to verify how a particular call is handled. This can be of value if you suspect DL/I is not operating correctly in a specific situation. You can issue the calls suspected of not executing properly and examine the results.



---

## Appendix C. The Database Resource Adapter (DRA)

The DRA is an interface to IMS DB full-function databases and data entry databases (DEDBs). The DRA can be used by a coordinator controller (CCTL) or a z/OS application program that uses the Open Database Access (ODBA) interface. This chapter is intended for the designer of a CCTL or an ODBA application program. If you want more information about a specific CCTL's interaction with IMS DB or DB/DC, see the documentation for that CCTL.

### **Related Reading:**

- For additional information on defining the ODBA interface, see *IMS Version 9: Installation Volume 2: System Definition and Tailoring*
- For information on designing application programs that use ODBA, see *IMS Version 9: Application Programming: Design Guide*

### **In This Chapter:**

- “Thread Concepts”
- “Sync Points” on page 358
- “The DRA Startup Table” on page 362
- “Enabling the DRA for a CCTL” on page 363
- “Processing CCTL DRA Requests” on page 365
- “Processing ODBA Calls” on page 366
- “CCTL-Initiated DRA Function Requests” on page 366
- “Terminating the DRA” on page 375
- “Designing the CCTL Recovery Process” on page 375
- “CCTL Performance—Monitoring DRA Thread TCBs” on page 376

---

## Thread Concepts

A *DRA thread* is a DRA structure that connects:

- A CCTL task (which makes database calls to IMS DB) with an IMS DB task that can process those calls. A CCTL thread is a CCTL task that issues IMS DB requests using the DRA.
- A z/OS application program task (which makes database calls to IMS DB) with an IMS DB task that can process those calls. An ODBA thread is a z/OS task that issues IMS DB calls using the DRA.

A single DRA thread is associated with every CCTL or ODBA thread. CCTL or ODBA threads cannot establish a connection with more than one DRA thread at a time.

## Processing Threads

The way that the DRA processes a CCTL thread is different from how it processes an ODBA thread.

### **Processing a CCTL Thread**

When a CCTL application program needs data from an IMS DB database, a CCTL task must issue a SCHED request for a PSB. To process the SCHED request, the DRA must create a DRA thread. To do this, the DRA chooses an available DRA thread TCB and assigns to it the CCTL thread token (a unique token that CCTL puts in the SCHED PAPL PAPLTTOK) and its own IMS DB task, which schedules

the PSB. If the scheduling is successful, the DRA thread is considered **complete** because it now connects a CCTL thread to a IMS DB task using a specific DRA thread TCB.

Subsequent DRA requests from this CCTL thread must use the same CCTL thread token in order to ensure that the request goes to the correct DRA thread. When the application program finishes and the CCTL thread no longer needs the services of the DRA thread, the CCTL issues a TERMTHRD (Terminate Thread) request to remove the CCTL thread token from the DRA thread TCB and terminates the DRA thread. The thread TCB can then become part of a new DRA thread.

### Processing an ODBA Thread

When an ODBA application program needs data from an IMS DB database, an ODBA task must issue an APSB call to initialize the ODBA environment. To process the APSB call, the DRA creates a DRA thread. The DRA chooses an available DRA thread TCB and assigns to it the ODBA thread and its own IMS DB task, which schedules the PSB. If the scheduling is successful, the DRA thread is considered **complete** because it now connects an ODBA thread to a IMS DB task using a specific DRA thread block.

When the application program finishes and the ODBA thread no longer needs the services of the DRA thread, the ODBA application issues a DPSB call to terminate the DRA thread. The thread block can then become part of a new DRA thread.

## Processing Multiple Threads

The DRA is capable of processing more than one thread at the same time. This is known as *multithreading*. Multithreading means that multiple CCTL or ODBA threads can be using the DRA at the same time. Multithreading applies to all DRA requests and ODBA calls.

### Processing Multiple CCTL Threads

To use the multithreading capability:

- The DRA must be initialized with more than one thread TCB. To initialize the DRA with more than one thread TCB, specify the MINTHRD and MAXTHRD parameters (in the DRA Startup Table) as greater than one.
- The CCTL must be capable of processing its CCTL threads concurrently.
- The CCTL must have Suspend and Resume exit routines. The DRA uses these routines to notify the CCTL of the status of thread processing.

### Processing Multiple ODBA Threads

To use the multithreading capability, the DRA must be initialized with more than one DRA thread. To do this, specify the MINTHRD and MAXTHRD parameters (in the DRA Startup Table) as greater than one.

## CCTL Multithread Example

Events in a multithreading system are shown in chronological order from top to bottom in Table 76 on page 357. To illustrate the concept of concurrent processing, the figure is split into two columns.

There are two CCTL threads and two DRA threads in the example. xxxRTNA is the module name (for this example) of the CCTL routine that builds PAPLs and calls DFSPRRC0 to process DRA requests.



Table 76. Example of Events in a Multithreading System

CCTL TCB Events	DRA TCB Events
<p>Application program1 needs a PSB, so CCTL thread1 is created. CCTL thread1 events:</p> <ul style="list-style-type: none"> <li>• DFSRTNA builds the SCHED PAPL and calls DFSPRRC0.</li> <li>• DFSPRRC0 creates a DRA thread, and the thread token (PAPLTTOK) is assigned to DRA thread TCB1.</li> <li>• DFSPRRC0 activates thread TCB1.</li> <li>• DFSPRRC0 calls the Suspend exit routine.</li> <li>• The Suspend exit routine suspends CCTL thread1.</li> </ul> <p>CCTL can now dispatch other CCTL threads for the CCTL TCB. Application program2 needs a PSB, so CCTL thread2 is created. CCTL thread2 events:</p> <ul style="list-style-type: none"> <li>• DFSRTNA builds the SCHED PAPL and calls DFSPRRC0.</li> <li>• DFSPRRC0 creates a DRA thread, and a new thread token (PAPLTTOK) is assigned to DRA thread TCB2.</li> <li>• DFSPRRC0 activates thread TCB2.</li> <li>• DFSPRRC0 calls the Suspend exit routine. The Suspend exit routine suspends CCTL thread2.</li> </ul> <p>Both threads are now suspended. The CCTL TCB is inactive until one of the threads resumes execution.</p> <p>Thread2 can resume immediately because the CCTL TCB is idle. It resumes execution directly after the point at which it was suspended by the Suspend exit routine.</p>	<p>DRA thread TCB1 events:</p> <ul style="list-style-type: none"> <li>• The DRA processes the SCHED request and asks IMS DB to perform a schedule process.</li> <li>• Scheduling is in progress.</li> </ul> <p>DRA thread TCB2 events:</p> <ul style="list-style-type: none"> <li>• The DRA processes the SCHED request and asks IMS DB to perform a schedule process.</li> <li>• Scheduling is in progress.</li> </ul> <p>TCB2 scheduling finishes first. DRA thread TCB2 events:</p> <ul style="list-style-type: none"> <li>• Scheduling completes in IMS DB, and the PAPL is filled in with the results.</li> <li>• The DRA calls the Resume exit routine and passes the PAPL back to the CCTL.</li> <li>• The Resume exit routine sees the thread token (PAPLTTOK) and flags CCTL thread2 as 'ready to resume'.</li> <li>• The Resume exit routine returns to the DRA, and TCB2 becomes inactive.</li> </ul> <p>TCB1 scheduling completes. DRA thread TCB1 events:</p>

Table 76. Example of Events in a Multithreading System (continued)

CCTL TCB Events	DRA TCB Events
<p>Thread1 must wait until the Resume exit routine is available because thread2 has just resumed.</p>	<ul style="list-style-type: none"> <li>• Scheduling completes in IMS DB and the PAPL is filled in with the results.</li> <li>• The DRA calls the Resume exit routine and passes the PAPL back to the CCTL.</li> <li>• The Resume exit routine sees the thread token (PAPLTTOK) and flags CCTL thread1 as 'ready to resume'.</li> <li>• The Resume exit routine returns control to the DRA and TCB1 becomes inactive.</li> </ul>
<p>CCTL thread2 events:</p> <ul style="list-style-type: none"> <li>• The Suspend exit routine returns to its caller, DFSPRRCO.</li> <li>• DFSPRRCO returns to DFSRTNA.</li> <li>• DFSRTNA gets the results from the SCHED PAPL and gives them to the application program2.</li> <li>• DFSRTNA finishes the thread2 SCHED request.</li> </ul>	
<p>After thread2 completes in CCTL TCB, the CCTL can dispatch thread1, which is currently waiting.</p>	
<p>CCTL thread1 events:</p> <ul style="list-style-type: none"> <li>• The Suspend exit routine returns to its caller, DFSPRRCO.</li> <li>• DFSPRRCO returns to DFSRTNA.</li> <li>• DFSRTNA gets the results from the SCHED PAPL and gives them to the application program1.</li> <li>• DFSRTNA finishes the thread1 SCHED request.</li> </ul>	

---

## Sync Points

*Sync point processing* finalizes changes to resources. Sync point requests specify actions to take place for the resource changed (for example, commit or abort). A sync point is when IMS DB actually processes the request.

Each sync point is based on a unit of recovery (UOR). A UOR covers the time during which database resources are allocated and can be updated until a request is received to commit or abort any changes. Normally, the UOR starts with a CCTL SCHED (schedule a PSB) request or an ODBA APSB call and ends with a sync point request. Other DRA thread requests can also define the start and end of a UOR.

A CCTL UOR is identified by a recovery token (PAPLRTOK) that is received as part of a thread request that creates a new UOR. It is 16 bytes in length. The first 8 bytes contain the CCTL identification. This identification is the same as the CCTL ID that was a final DRA startup parameter determined from USERID or PAPLUSID in INIT request. The second 8 bytes must be a unique identifier specified by the CCTL for each UOR.

**Related Reading:** See the request descriptions under “CCTL-Initiated DRA Function Requests” on page 366 for more information on the DRA thread requests.

IMS DB expects the CCTL or the ODBA application to make the sync point decision and the resulting request. In the case of a CCTL, the CCTL is the sync point manager and coordinates the sync point process with all of the database resource managers (including those other than IMS DB) that are associated with a UOR. In the case of an ODBA application, RRS/MVS is the sync point manager and coordinates all the resource managers (including those other than IMS) that are associated with the UOR.

A CCTL working with a single resource manager may request a sync point in a single request or can use the two-phase sync point protocol which is required for a CCTL working with multiple resource managers. The single-phase sync point request can be issued when the CCTL has decided to commit the UOR, and when IMS DB owns all of the resources modified by the UOR.

An ODBA application must use the two-phase sync point protocol for committing changes to the IMS database.

## The Two-Phase Commit Protocol

The two-phase sync point protocol consists of two requests issued by the sync point manager to each of the resource managers involved in the UOR:

- Phase 1**            The sync point manager asks all participants if they are ready to commit a UOR.
- Phase 2**            The sync point manager tells each participant to commit or abort based on the response to the request issued in phase 1.

A UOR has two states: in-flight and in-doubt. The UOR is in an in-flight state from its creation time until the time IMS DB logs the phase 1 end (point C in Table 77 on page 360 and Table 78 on page 360). The UOR is in an in-doubt state from (point C) until IMS DB logs phase 2 (point D in Table 77 on page 360 and point H in Table 78 on page 360).

The in-doubt state for a single-phase sync point request is a momentary state between points C and D in Table 77 on page 360.

The in-flight and in-doubt states are important because they define what happens to the UOR in the event of a thread failure. If a thread fails while its IMS DB UOR is in-flight the UOR database changes are backed out. If a thread fails when its IMS DB UOR is in-doubt, during single-phase commit, the UOR database changes are kept for an individual thread abend, but are not kept for a system abend. If a thread fails when its IMS DB UOR is in-doubt during two-phase commit, the database changes are kept.

Thread failure refers to either of these cases:

- Individual thread abends.
- System abends: IMS DB failure, CCTL failure, ODBA application failure, or z/OS failure (which abends all threads).

The following figure shows the system events that occur when CCTL is used for single-phase sync point processing.

Time →  
 —A—B—C—D—E—

Table 77. CCTL Single-Phase Sync Point Processing

Points In Time	System Events
A	CCTL phase 1 send
B	IMS DB phase 1 receive
C	IMS DB log phase 1 end
D	IMS DB log phase 2
E	CCTL phase 2 receive

Table 78 below shows the system events that occur when CCTL is used for two-phase sync point processing.

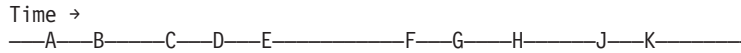


Table 78. CCTL Two-Phase Sync Point Processing

Points In Time	System Events
A	CCTL phase 1 send
B	IMS DB phase 1 receive
C	IMS DB log phase 1 end
D	IMS DB phase 1 respond
E	CCTL phase 1 receive
F	CCTL phase 2 send
G	IMS DB phase 2 receive
H	IMS DB log phase 2
J	IMS DB phase 2 respond
K	CCTL phase 2 receive

The following figure shows the system events that occur when two-phase sync point processing is done using ODBA.

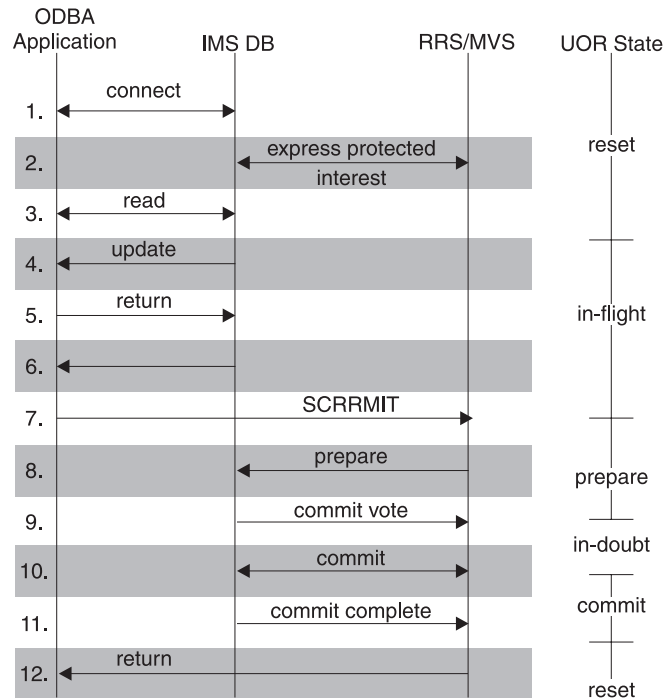


Figure 75. ODBA Two-Phase Sync Point Processing

**Notes:**

1. The ODBA application and IMS DB make a connection using the ODBA interface.
2. IMS expresses protected interest in the work started by the ODBA application. This informs RRS/MVS that IMS will participate in the two-phase commit process.
3. The ODBA application makes a read request to an IMS resource.
4. The ODBA application updates a protected resource.
5. Control is returned to the ODBA application following its update request.
6. The ODBA application requests that the update be made permanent by issuing the SRRRCMIT call.
7. RRS/MVS calls IMS to do the prepare (phase 1) process.
8. IMS returns to RRS/MVS with its vote to commit.
9. RRS/MVS calls IMS to do the commit (phase 2) process.
10. IMS informs RRS/MVS that it has completed phase 2.
11. Control is returned to the ODBA application following its commit request.

**In-Doubt State During Two-Phase Sync**

A IMS DB UOR remains in the in-doubt state until a phase 2 request is received. This process is called “resolving the in-doubt”. While a UOR is in-doubt, the database resources owned by that UOR are inaccessible to other requests. It is vital that in-doubts are resolved immediately.

**CCTL Example:** If in-doubt UORs are created because IMS DB failed, the following sequence must occur to resolve the in-doubt UORs.

1. After restarting IMS DB, the CCTL should identify itself to IMS DB using an INIT request.

2. If identification is successful, the DRA notifies the CCTL control exit, passing to it a list of IMS DB UORs that are in-doubt.
3. The CCTL must resolve each in-doubt by making a RESYNC call, which causes a phase 2 action, commit or abort.

For CCTL to resolve a IMS DB in-doubt UOR, the CCTL must have a record of this UOR and the appropriate phase 2 action it must take. In this example, the CCTL record of a possible IMS DB in-doubt UOR is called a transition UOR.

4. The CCTL must define a transition UOR for the interval A-K (refer to Table 78 on page 360). Because this interval encompasses the IMS DB in-doubt period C-H, CCTL can resolve any in-doubts.

If a CCTL defines a transition UOR as interval E-K, the following problem can arise: If IMS DB fails while a thread is between C and D, IMS DB has an in-doubt UOR for which CCTL has no corresponding transition UOR, even though the phase 1 call failed. CCTL cannot resolve this UOR during the identify process. The only way to resolve this in-doubt is by using the IMS DB command, CHANGE INDOUBT.

**ODBA Example:** For ODBA, all in-doubts are resolved through z/OS using the Recoverable Resource Service (RRS).

---

## The DRA Startup Table

The DRA Startup Table contains values used to define the characteristics of the DRA. The DRA Startup Table is created by assembling:

- The DFSPZPxx module for a CCTL's use.
- The DFSxxxx0 module for ODBA's use.

The CCTL or ODBA system programmer must make the required changes to these modules to correctly specify the DRA parameters desired. The DRA parameters are specified as keywords on the DFSPRP macro invocation. These keywords and their meanings are listed following the sample DFSPZP00 source code.

## Sample DFSPZP00 Source Code:

```
DFSPZP00 CSECT
          DFSPRP DSECT=NO
          END
```

## DFSPRP Macro Keywords

Keyword	Description
<b>AGN=</b>	A one-to-eight character application group name. This is used as part of the IMS DB and DB/DC security function (see <i>IMS Version 9: Administration Guide: System</i> for more information on IMS DB and DB/DC security).
<b>CNBA=</b>	Total Fast Path NBA buffers for the CCTL's or ODBA's use. For a description of Fast Path DEDB buffer usage, see <i>IMS Version 9: Administration Guide: System</i> .
<b>DBCTLID=</b>	The four-character name of the IMS DB or DB/DC region. This is the same as the IMSID parameter in the DBC procedure. For more information on the DBC procedure, see <i>IMS Version 9: Installation Volume 2: System Definition and Tailoring</i> . The default name is SYS1.

<b>DDNAME=</b>	A one-to-eight character ddname used with the dynamic allocation of the IMS DB execution library. The default ddname is CCTLDD.
<b>DSNAME=</b>	A 1-to-44 character data set name of the IMS DB execution library, which must contain the DRA modules and must be z/OS authorized. The default DSNAME is IMS.SDFSRESL. This library must contain the DRA modules.
<b>FPBOF=</b>	The number of Fast Path DEDB overflow buffers allocated per thread. For a description of Fast Path DEDB buffer usage, see <i>IMS Version 9: Administration Guide: System</i> . The default is 00.
<b>FPBUF=</b>	The number of Fast Path DEDB buffers allocated and fixed per thread. For a description of Fast Path DEDB buffer usage, see <i>IMS Version 9: Administration Guide: System</i> . The default is 00.
<b>FUNCLV=</b>	Specifies the DRA level that the CCTL or ODBA supports. The default is 1.
<b>IDRETRY=</b>	The number of times a z/OS application region is to attempt to IDENTIFY (or attach) to IMS after the first IDENTIFY attempt fails. The maximum number 255. The default is 0.
<b>MAXTHRD=</b>	The maximum number of DRA thread TCBs available at one time. The maximum number is 999. The default is number 1.
<b>MINTHRD=</b>	The minimum number of DRA thread TCBs to be available at one time. The maximum number is 999. The default is number 1.
<b>SOD=</b>	The output class used for a SNAP DUMP of abnormal thread terminations. The default is A.
<b>TIMEOUT=</b>	(CCTL only). The amount of time (in seconds) a CCTL waits for the successful completion of a DRA TERM request. Specify this value only if the CCTL application is coded to use it. This value is returned to the CCTL upon completion of an INIT request.
<b>TIMER=</b>	The time (in seconds) between attempts of the DRA to identify itself to IMS DB or DB/DC during an INIT request. The default is 60 seconds.
<b>USERID=</b>	An eight-character name of the CCTL or ODBA region. This keyword is ignored for an ODBA Region.

---

## Enabling the DRA for a CCTL

This section describes the two steps required to enable the DRA.

1. The CCTL system programmer must copy the DRA Startup/Router routine (DFSPRRC0) into a CCTL load library, because the CCTL must load DFSPRRC0. Although the DRA is shipped with the IMS product, it runs in the CCTL address space.

The system programmer can copy the routine from the IMS.SDFSRESL library (built by the IMS generation process), or can concatenate the IMS.SDFSRESL library to the ODBA step library.

2. The system programmer must put the DFSPZPxx load module (DRA Startup Table) in a load library. The DRA is now ready to be initialized.

## Initializing the DRA

The CCTL starts the initialization process as a result of the CCTL application program issuing an initialization (INIT) request. At this point in time, the CCTL loads DFSPRRC0 and then calls the DRA to process the INIT request.

As part of the initialization request, the CCTL application program specifies the startup table name suffix (xx). The default load module, DFSPZP00, is in the IMS.SDFSRESL library.

After processing the INIT request, the DRA identifies itself to IMS DB. The DRA is then capable of handling other requests.

**Related Reading:** For an example of DFSPZP00, see *IMS Version 9: Installation Volume 2: System Definition and Tailoring*.

DFSPZP00 contains default values for the DRA initialization parameters. If you want to specify values other than the defaults, write your own module (naming it DFSPZPxx), assemble it, and load it in the CCTL load library. Use the supplied module, DFSPZP00, as an example.

The remainder of the DRA modules reside in a load library that is dynamically allocated by DFSPRRC0. The DDNAME and DSNAME of this load library are specified in the startup table. The default DSNAME (IMS.SDFSRESL) contains all the DRA code and is specified in the default startup table, DFSPZP00.

---

## Enabling the DRA for the ODBA Interface

There are four steps required to enable the DRA before an ODBA interface can use it:

1. Create the ODBA DRA Startup Table.
2. Verify that the ODBA and DRA modules reside in the STEPLIB or JOBLIB in the z/OS application region.
3. Link the ODBA application programs with DFSCDLI0.
4. Set up security.

These steps are described in detail in the *IMS Version 9: Installation Volume 2: System Definition and Tailoring*.

## Initializing the DRA

The ODBA interface starts the initialization process as a result of the ODBA application program issuing an initialization (CIMS INIT) request or an APSB call. At this point in time, the ODBA interface calls the DRA to process the CIMS INIT request or APSB call. Optionally, the ODBA application program can specify the startup table name (xxxx) in the AIBRSNM2 field of the AIB.

After processing the CIMS INIT request, the DRA identifies itself to one IMS DB. The DRA is then capable of handling other requests. The DRA's structure at this time is shown in Figure 76 on page 365.



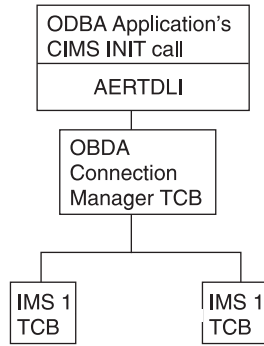


Figure 76. DRA Component Structure with the ODBA Interface

The remainder of the DRA modules reside in a load library that is dynamically allocated by DFSAERA0. The DDNAME and DSNAME of this load library are specified in the startup table. The default DSNAME (IMS.SDFSRESL) contains all the DRA code.

## Processing CCTL DRA Requests

The CCTL communicates with IMS DB through DRA requests. These requests are passed from the CCTL to the DRA using a participant adapter parameter list (PAPL). See Appendix B for a sample PAPL listing. There are different types of DRA requests shown in the sample PAPL listing.

To make a DRA request the CCTL must pass control to the DRA Startup/Router Routine DFSPRRC0, and have register 1 point to a PAPL.

Before passing control to DFSPRRC0, the CCTL must fill in the PAPL according to the desired request. These requests are specified by a function code in the PAPLFUNC field.

To specify a thread function request, put the PAPLTFUN value into the PAPLFUNC field.

The function requests are further broken down into many subfunctions. A thread function request is referred to by its subfunction name (for example, a thread request with a schedule subfunction is referred to as a SCHED request). Non-thread function requests are referred to by function name (for example, an initialization request is called an INIT request).

The term “DRA request” applies to both thread and non-thread function requests.

Once the PAPL is built and the DRA Startup/Router routine is loaded, the CCTL passes control to DFSPRRC0. The contents of the registers upon entry to DFSPRRC0 are:

Register	Contents
1	Address of the PAPL
13	Address of a standard 18-word save area
14	Return address of the calling routine

The DRA Startup/Router routine puts itself into 31-bit addressing mode and will return to the calling routine in the caller's original addressing mode with all its registers restored. Register 15 is always returned with a zero in it.

The return code for the request is in the PAPLRETC field of the PAPL.

---

## Processing ODBA Calls

Unlike a CCTL's use of the PAPL, an ODBA application program communicates with IMS DB using the AERTDLI interface. The AERTDLI call interface processes DL/I calls from the ODBA application and also returns the results of those calls back to the ODBA using an AIB.

**Related Reading:** For information on using the AIB mask for configuring ODBA calls, see "Specifying the AIB Mask for ODBA Applications" on page 99.

---

## CCTL-Initiated DRA Function Requests

This section documents General-Use Programming Interface and Associated Guidance Information.

This section discusses the requests available to the CCTL that allow it to communicate with DBCTL. These requests are passed to the DRA through the PAPL.

For all DRA requests, there are PAPL fields that the CCTL must fill in. When the DRA completes the request, there are some output PAPL fields that the DRA fills in. Some fields in the returned PAPL might contain the original input value.

(The PAPLTOK and PAPLUSER fields **will** retain the original input values.)

The PAPLUSER field is a field to be used at the CCTL's discretion. One possible use for it is to pass data to exit routines.

The DRA returns a code (in the PAPLRETC field) to the CCTL after processing a DRA request. The code indicates the status of the request and can be either an IMS code, a DRA code, or a z/OS code. Failed DRA requests return a nonzero value in the PAPLRETC field.

**Related Reading:**

- See "Problem Determination" on page 380 for more information on the codes returned when a DRA request fails.
- For a complete list and description of all DRA return codes, see *IMS Version 9: Messages and Codes, Volume 1*.

## INIT Request

The INIT request initializes the DRA. The DRA startup parameter table contains all of the required parameters that you need to define the DRA. You can use the parameters given in the default module, DFSPZP00, or you can write your own module and link-edit it into the IMS.SDFSRESL.

**Related Reading:** For more information, see "Enabling the DRA for a CCTL" on page 363.

The INIT PAPL also contains some parameters needed to initialize the DRA. If the same parameter appears in both the INIT PAPL and in the DRA startup parameter table, the specification in the INIT PAPL will override that in the startup table.

In addition to the required parameters, you can also include the following optional parameters in the INIT PAPL:

Field	Contents
<b>PAPLFUNC</b>	PAPLINIT
<b>PAPLSUSP</b>	The address of the Suspend exit routine
<b>PAPLRESM</b>	The address of the Resume exit routine
<b>PAPLCNTL</b>	The address of the Control exit routine
<b>PAPLTSTX</b>	The address of the Status exit routine

After the INIT request and the startup table have been processed, the DRA returns the following data to the CCTL in the INIT PAPL:

Field	Contents
<b>PAPLDBCT</b>	The IMS DB identifier (this is the IMSID parameter from system generation)
<b>PAPLCTOK</b>	The request token that identifies the CCTL to the DRA
<b>PAPLTIMO</b>	DRA TERM request timeout value (in seconds)
<b>PAPLRETC</b>	A code returned to the CCTL specifying the status of the request
<b>PAPLDLEV</b>	A flag indicating to CCTL which functions the DRA supports. (For the latest version of PAPL mapping format see the IMS. library; member name is DFSPAPL.)

### INIT Request, Identify to DBCTL

To make the DRA functional, the DRA must identify itself to IMS DB, thus establishing a link between IMS DB and the CCTL. The identify process occurs in two cases:

- As a direct result of an INIT request.
- As part of a terminate/reidentify request from a Control exit routine invocation.

The DRA identifies itself to the IMS DB subsystem specified in the final DRA startup parameters. The identify process executes asynchronously to the INIT process. Therefore, it is possible for the INIT request to complete successfully while the identify process fails. In this case, the Control exit routine notifies the CCTL that the connection to IMS DB failed.

If IMS DB is not active, the console operator will receive a DFS690 message (a code of 0 was returned in the PAPLRETC field). You must reply with either a CANCEL or WAIT response. If you reply with WAIT, the DRA waits for a specified time interval before attempting to identify again. The waiting period is necessary because the identify process won't succeed until the DBCTL restart process is complete. You specify the length of the waiting period on the TIMER DRA startup parameter. If subsequent attempts to identify fail, the console operator will receive message DFS691, WAITING FOR IMS DB.

If the DRA cannot identify to IMS DB because the subsystem does not reach a restart complete state, there are two ways to terminate the identify process:

- The Control exit routine is called with each identify failure. This sets a PAPL return code of 4 or 8, which terminates the identify process.
- The CCTL can issue a TERM request.

If you reply with CANCEL to message DFS690, control is passed to the Control exit routine, and the DRA acts upon the routine's decision.

After the identify process successfully completes, the DRA makes the CCTL address space non-swappable and calls the Control exit routine with a list of in-doubt UORs. If no in-doubt UORs exist, a null list is passed. The CCTL can use the RESYNC request to resolve any in-doubt UORs that do exist.

The INIT request will attempt to create the MINTHRD number of thread TCBs. The actual number of TCBs created might be less than this value due to storage constraints.

### INIT Request after a Previous DRA Session Termination

If a prior DRA session ended with a TERM request that received a PAPL return code=0, this INIT request must specify PAPLCTOK=0. If PAPLCTOK other than 0 is sent, the INIT request will fail.

The INIT request must pass the prior session's PAPLCTOK value in the current PAPLCTOK field if a DRA session ended one of the following ways:

- A nonzero return code from a TERM request.
- An internal TERM request from a Control exit routine request.
- A DRA failure.

## RESYNC Request

The RESYNC request tells IMS DB what to do with in-doubt UORs. The following 4 subfunction values indicate possible actions:

<b>PAPLRCOM</b>	Commit the in-doubt UOR.
<b>PAPLRABT</b>	Abort the in-doubt UOR. Changes made to any recoverable resource are backed out.
<b>PAPLSCLD</b>	The UOR was lost to the transaction manager due to a coldstart.
<b>PAPLSUNK</b>	The in-doubt UOR is unknown to the CCTL. This can occur when the CCTL's in-doubt period does not include the start of phase 1. (See Table 78 on page 360 for an illustration of in-doubt periods.)

You must fill in the following input fields of the PAPL:

Field	Contents
<b>PAPLCTOK</b>	Request token  This token identifies the CCTL to the DRA. The DRA establishes the token and returns it to the CCTL in the parameter list on the startup INIT request. The request token must be passed on to the DRA for all RESYNC requests.
<b>PAPLR TOK</b>	Recovery token  This 16-byte token is associated with a UOR. The first 8 bytes must be the transaction manager subsystem ID. The second 8 bytes must be unique for one CCTL thread. This is one of the in-doubt recovery tokens passed to the Control exit routine.

**PAPLFUNC** PAPLSYN  
**PAPLSNC** One of the four values listed above

## TERM Request

The TERM request results in a termination of the IMS DB/CCTL connection and a removal of the DRA from the CCTL environment. The DRA terminates after all threads have been resolved. No new DRA or thread requests are allowed, and current requests in progress must complete.

You must fill in the following input fields in the PAPL:

Field	Contents
<b>PAPLFUNC</b>	PAPLTERM, DRA terminate function code
<b>PAPLCTOK</b>	The DRA request token (output from an INIT request)

After receiving the TERM request results, the CCTL may remove DFSPPRC0.

The following output fields are returned in the PAPL to the CCTL:

Field	Contents
<b>PAPLRETC</b>	The return code
<b>PAPLMXNB</b>	The number of times the maximum thread count was encountered during this DRA session
<b>PAPLMTNB</b>	The number of times the minimum thread count was encountered during this DRA session
<b>PAPLHITH</b>	The largest number of thread TCBs that were scheduled during this DRA session
<b>PAPLTIMX</b>	The elapsed time at maximum thread for this DRA session

## Thread Function Requests

The Thread Function requests consist of the SCHED, IMS, SYNTerm, PREP, COMTERM, ABTTerm, and TERMTHRD requests and are described in this section.

### SCHED Request

The SCHED request schedules a PSB in IMS DB. The first SCHED request made by a CCTL thread requires a new DRA thread. If any existing DRA thread TCBs are not currently processing a DRA thread, one of these is used. If no TCBs are available, the DRA either creates a new thread TCB (until the maximum number of threads as specified by the MAXTHRD parameter in the INIT request is reached), or makes the SCHED request wait until a thread becomes available.

The value in the PAPLWCMD field indicates whether the thread to which the SCHED request applies is a short or long thread. The type of thread determines the action that IMS takes when a database command is entered for a database scheduled to the thread. The /STOP DATABASE, /DBDUMP DATABASE, or /DBRECOVERY DATABASE command issued against a database scheduled on a short thread will wait for the database to be unscheduled. IMS rejects these commands if they are entered for a database scheduled on a long thread.

You must fill in the following input fields in the PAPL:

Field	Contents
-------	----------

<b>PAPLFUNC</b>	PAPLTFUN, thread function code
<b>PAPLSFNC</b>	PAPLSCHE, schedule request subfunction code
<b>PAPLCTOK</b>	The DRA request token (output from an INIT request)
<b>PAPLTTOK</b>	The thread token set up by the CCTL
<b>PAPLR TOK</b>	The 16-byte UOR token (RTOKEN). For more information about UORs, see “Sync Points” on page 358.
<b>PAPLPSB</b>	The PSB name
<b>PAPLWRTH</b>	Deadlock Worth Value  If this thread hits a deadlock condition with any other DRA thread or with any IMS region, DBCTL collapses the thread with the lower deadlock worth value.
<b>PAPLWCMD</b>	This bit defines the thread as either a short or long thread which determines what action IMS takes on a /STOP DATABASE, /DBDUMP DATABASE, or /DBRECOVERY DATABASE command for a database scheduled to the thread. If the bit is set on (X'80'), the database is scheduled on a short thread; if the bit is set off, the database is scheduled for a long thread.
<b>PAPLFTRD</b>	Fast Path Trace Option  If this bit is on (X'40'), Fast Path tracing in IMS DB is activated. (For more information, see “Tracing” on page 379 in “CCTL Performance—Monitoring DRA Thread TCBs” on page 376.)
<b>PAPLKEYP</b>	Public Key Option  If this bit is set (X'10'), DBCTL will build UPSTOR area in a special subpool so that applications running in public key can fetch the UPSTOR area.
<b>PAPLLKGV</b>	Lockmax Option  If this bit is set (X'08'), DBCTL uses the value in PAPLLKMX as the maximum number of locks that this UOR can hold. Exceeding the maximum results in a U3301 abend.
<b>PAPLLKMX</b>	Lockmax Value, 0 to 255  This value overrides any LOCKMAX parameter specified on the PSBGEN for the PSB referenced in the SCHED request.
<b>PAPLALAN</b>	Application language type

Specifying the following input field is optional:

<b>Field</b>	<b>Contents</b>
<b>PAPLSTAT</b>	Address of an area where scheduled statistical data is returned to the CCTL.
<b>PAPLPBTK</b>	Address of the token for the z/OS Workload Manager performance block obtained by the CCTL.  You must specify this field for z/OS Workload Manager support for DRA threads.

The following output fields are returned in the PAPL to the CCTL:

<b>Field</b>	<b>Contents</b>
--------------	-----------------

<b>PAPLRETC</b>	The return code
<b>PAPLCTK2</b>	The thread request token number 2. This is another DRA token required on future DRA requests originating from this thread.
<b>PAPLPCBL</b>	The address of the PCB list. There is one entry in the list for each PCB in the PSB that was scheduled, even if the PCB cannot be used with IMS DB.
<b>PAPL1PCB</b>	The address of the PCBLIST entry pointing to the first database PCB
<b>PAPLIOSZ</b>	The size of the maximum I/O area
<b>PAPLPLAN</b>	The language type of the PSB
<b>PAPLMKEY</b>	The maximum key length
<b>PAPLSTAT</b>	The address of the schedule statistical data area. This address must be specified on the input field.

CCTLs currently using the IMS Database Manager and migrating to DBCTL will experience a change in the PCBLIST and user PCB area on a schedule request. The first PCB pointer in the PCBLIST contains the address of an I/O PCB. The I/O PCB is internally allocated during the schedule process in a DBCTL environment. The I/O PCB is normally used for output messages or to request control type functions to be processed. The PCBLIST and the PCBs reside in a contiguous storage area known as UPSTOR. If the PSB was generated with LANG=PLI, the PCBLIST points to pointers for the PCBs. If LANG= was not PLI, the PCBLIST points to the PCBs directly.

### IMS Request

This request makes an IMS or Fast Path database request against the currently scheduled PSB.

You must fill in the following input fields in the PAPL:

Field	Contents
<b>PAPLFUNC</b>	PAPLTFUN
<b>PAPLSFNC</b>	PAPLDLI, DL1 request subfunction code
<b>PAPLCTOK</b>	DRA request token (output from an INIT request)
<b>PAPLCTK2</b>	Thread Token number 2. This is the DRA request token that is part of the output from a SCHED request.
<b>PAPLTTOK</b>	Thread token set up by the CCTL
<b>PAPLR TOK</b>	RTOKEN  A 16-byte UOR token. See "Sync Points" on page 358 for more information about UORs.
<b>PAPLCLST</b>	The address of an IMS call list. See "Chapter 2, Defining Application Program Elements" for call list formats.
<b>PAPLALAN</b>	Application language type. This must reflect how the call list is set up. If PAPLALAN='PLI', the DRA expects the call list to contain pointers to the PCB's pointers. For any other programming language, the DRA expects direct pointers.  PAPLALAN does not have to match PAPLPLAN which schedules request returns. For example, if PAPLPLAN=PLI, the PCBLIST in

UPSTOR points to an indirect list. If desired, the CCTL can use this to create a PCBLIST that application programs use. If the application programs are written in COBOL, the CCTL may create a new PCBLIST without pointers as long as the new list actually points to PCBs in UPSTOR. The application program IMS call lists can specify PAMPLALAN=COBOL, and the DRA will not expect pointers in the call list.

The following output fields are returned in the PAMPL to the CCTL:

Field	Contents
PAPLRETC	Code returned
PAPLSEGL	Length of data returned

### SYNTERM Request

This is a single-phase sync point request to commit the UOR. It also releases the PSB.

You must fill in the following input fields in the PAMPL:

Field	Contents
PAPLFUNC	PAPLTFUN
PAPLSFNC	PAPLSTRM, sync point commit/terminate subfunction code
PAPLCTOK	DRA request token (output from INIT request)
PAPLCTK2	The thread request token number 2. This DRA token is the output from the SCHED request.
PAPLTTOK	The thread token set up by the CCTL
PAPLR TOK	A 16-byte UOR token (RTOKEN). For information on UORs see "Sync Points" on page 358.

You can also specify the following, optional input fields:

Field	Contents
PAPLSTAT	Address of an area where transaction statistical data is returned to the CCTL.

The following output fields are returned in the PAMPL to the CCTL:

Field	Contents
PAPLRETC	Code returned
PAPLSSCC	State of the single-phase sync point request at the time of the thread failure. This field is set if PAPLRETC is not equal to zero.
PAPLSTAT	The address of the transaction statistical data area. The address must be specified on the input field.

### PREP Request

This is a phase 1 sync-point request that asks IMS DB if it is ready to commit this UOR.

You must fill in the following input fields of the PAMPL:

Field	Contents
-------	----------



<b>PAPLFUNC</b>	PAPLTFUN
<b>PAPLSFNC</b>	PAPLPREP, sync-point prepare subfunction code
<b>PAPLCTOK</b>	DRA request token (output from an INIT request)
<b>PAPLCTK2</b>	Thread Token number 2. This is the DRA request token which is output from a SCHED request.
<b>PAPLTTOK</b>	The thread token set up by the CCTL
<b>PAPLRTOK</b>	A 16-byte UOR token (RTOKEN). See “Sync Points” on page 358 for more information about UORs.

The following output fields are returned in the PAPL to the CCTL:

<b>Field</b>	<b>Contents</b>
<b>PAPLRETC</b>	Code returned
<b>PAPLSTCD</b>	Fast Path status code
	If the value in the PAPLRETC field is decimal 35, the PAPLSTCD field contains a status code that further describes the error.

### **COMTERM Request**

This is a phase 2 sync-point request to commit the UOR. It also releases the PSB. You must issue a PREP request prior to issuing a COMTERM request.

You must fill in the following input fields in the PAPL:

<b>Field</b>	<b>Contents</b>
<b>PAPLFUNC</b>	PAPLTFUN
<b>PAPLSFNC</b>	PAPLCTRM, sync-point commit/terminate subfunction code
<b>PAPLCTOK</b>	DRA request token (output from an INIT request)
<b>PAPLCTK2</b>	Thread Token number 2. This is the DRA request token, which is output from a SCHED request.
<b>PAPLTTOK</b>	The thread token set up by the CCTL
<b>PAPLRTOK</b>	A 16-byte UOR token (RTOKEN). See “Sync Points” on page 358 for more information about UORs.

Specifying the following input field is optional:

<b>Field</b>	<b>Contents</b>
<b>PAPLSTAT</b>	Address of an area where transaction statistical data is returned to the CCTL

The following output fields are returned in the PAPL to the CCTL:

<b>Field</b>	<b>Contents</b>
<b>PAPLRETC</b>	Code returned
<b>PAPLSTAT</b>	The address of the transaction statistical data area. This address must be specified on the input field.

### **ABTTERM Request**

This is a phase 2 sync-point request for abort processing. It also releases the PSB. It does not require a preceding PREP request.

You must fill in the following input fields of the PAPL:

Field	Contents
<b>PAPLFUNC</b>	PAPLTFUN
<b>PAPLSFNC</b>	PAPLATRM, sync-point abort/terminate subfunction code
<b>PAPLCTOK</b>	DRA request token (output from an INIT request)
<b>PAPLCTK2</b>	Thread Token number 2. This is the DRA request token, which is output from a SCHED request.
<b>PAPLTTOK</b>	The thread token set up by the CCTL
<b>PAPLRTOK</b>	A 16-byte UOR token (RTOKEN). See “Sync Points” on page 358 for more information about UORs.

Specifying the following input field is optional:

Field	Contents
<b>PAPLSTAT</b>	Address of an area where transaction statistical data is returned to the CCTL.

The following output fields are returned in the PAPL to the CCTL:

Field	Contents
<b>PAPLRETC</b>	Code returned
<b>PAPLSTAT</b>	The address of the transaction statistical data area. This address must be specified on the input field.

### **TERMTHRD (PAPLSFNC - PAPLTTHD) Request**

This request terminates the DRA thread.

You must fill in the following input fields of the PAPL:

Field	Contents
<b>PAPLFUNC</b>	PAPLTFUN
<b>PAPLSFNC</b>	PAPLTTHD, thread terminate subfunction code
<b>PAPLCTOK</b>	DRA request token (output from an INIT request)
<b>PAPLCTK2</b>	Thread Token number 2. This is the DRA request token which is output from a SCHED request.
<b>PAPLTTOK</b>	The thread token set up by the CCTL

Specifying the following input field is optional:

Field	Contents
<b>PAPLSTAT</b>	Address of an area where transaction statistical data is returned to the CCTL

The following output fields are returned in the PAPL to the CCTL:

Field	Contents
<b>PAPLRETC</b>	Code returned
<b>PAPLSTAT</b>	The address of the transaction statistical data area. This address must be specified on the input field.

---

## PAPL Mapping Format

The PAPL is the parameter list used by the DRA interface in a CCTL environment. For the latest version of PAPL mapping format, see the IMS.ADFSMAC library; the member name is DFSPAPL.

---

## Terminating the DRA

Termination isolation should be one of your primary considerations when you design a CCTL subsystem or an ODBA application.

**Definition:** *Termination isolation* means that a failure of the IMS DB subsystem does not cause a direct failure of any attached CCTL subsystem or ODBA application and vice versa.

Although IMS DB was designed to prevent failure between connecting subsystems, a termination of a CCTL subsystem can cause IMS DB failure. If a DRA thread TCB terminates while IMS DB is processing a thread DL/I call on the CCTL's behalf, IMS DB fails with a U0113 abend. To promote termination isolation, see the "Summary of CCTL Design Recommendations" on page 376.

The following conditions cause a thread TCB to terminate while IMS DB processes a DL/I call:

- A DRA thread abend due to code failure. This can be corrected by fixing the failing code.
- The CCTL TCB collapses while a thread TCB still exists. The thread TCB collapses with an S13E or S33E abend and can result from three situations: a CCTL abend, a cancel command, or a shutdown. The number of U0113 abends caused by a CCTL cancel command can be reduced by following the design recommendations listed in the "Summary of CCTL Design Recommendations" on page 376.
- A DRA thread abend due to a IMS DB /STOP REGION CANCEL command initiated by CCTL.

An IMS DB U0113 abend can be prevented by designing the CCTL recovery process so that it issues a TERM request and waits for the request to complete. This allows the DRA and thread TCBs to terminate before the CCTL TCB terminates.

---

## Designing the CCTL Recovery Process

Under the conditions of a nonrecoverable z/OS abend, a DRA TERM request lets all threads collapse and U0113 is possible. To reduce the number of nonrecoverable abends of the CCTL, IMS DB intercepts any operator CANCEL of a CCTL that is connected to IMS DB, and converts it to a S08E recoverable abend of the CCTL.

You can also as a last resort, force a CCTL to shut down. If an operator enters a FORCE command after CANCEL has been entered (and converted to S08E), IMS DB converts FORCE into an z/OS cancel command. Subsequent FORCE attempts are not intercepted by IMS DB. In these cases of nonrecoverable abends, a U0113 is possible.

A CCTL might have a means of allowing its own shutdown. The CCTL shutdown logic should issue a DRA TERM request and wait for the request completion to prevent a U0113 abend in IMS DB. The DRA TERM request waits for current thread

requests to complete. One thing that can prevent a current thread DL/I call from completing normally is if the call has to wait in IMS DB for a database segment to become available. The reason the segment might not be available is that it is held by another UOR, either in a thread belonging to another CCTL or in an IMS dependent region (for example, a BMP). The solution is to not have CCTL threads or BMPs that have long-running UORs.

**Recommendation:** BMPs should take frequent checkpoints.

No matter how you choose to prevent or discourage long-running CCTL threads, you must decide how long to wait for the DRA TERM request to complete (TIMEOUT). In most cases, it is undesirable to get a U113 abend in IMS DB during a CCTL termination, so the timeout value should be greater than the longest possible UOR. If the CCTL has a means of limiting the UOR time or allowing the installation to specify this time limit, the DRA TERM timeout value can be determined. This timeout value can be specified in the DRA startup table and is returned to the CCTL in the INIT PAPL.

**Recommendation:** CCTL should use this DRA TERM timeout value when waiting for the DRA TERM request to complete. At the very least, by using the DRA TERM timeout value, you can control whether CCTL terminations cause IMS DB failures with respect to the UOR time length of the applications that run in a given IMS DB/CCTL session.

## Summary of CCTL Design Recommendations

CCTL Operations Recommendation:

- Avoid using CANCEL or FORCE commands against CCTL regions that are connected to IMS DB.

CCTL Design Recommendations:

- The CCTL should issue a DRA TERM request during recoverable abend processing.
- CCTL shutdown functions should issue a DRA TERM request.
- Whenever a DRA TERM request is issued, wait for it to complete. If this time must have an upper limit, use the TIMEOUT value specified in the DRA startup table.
- The CCTL should prevent long-running UORs in its threads using IMS DB.

User Installation Recommendations:

- Have BMPs take frequent checkpoints.
- Limit long-running UOR applications.
- Set the TIMEOUT startup parameter as high as possible, preferably longer than longest running UOR.

---

## CCTL Performance—Monitoring DRA Thread TCBs

**Requirement:** The DRA initialization process requires a minimum and maximum value (MINTHRD and MAXTHRD) for DRA thread TCBs. The value of MINTHRD and MAXTHRD determine the number of multithreading executions that can occur concurrently. These values also define the range of thread TCBs that the DRA will maintain under normal conditions with no thread failures. The number of TCBs can go below the MINTHRD value when the following thread failures occur:

- An abend.

- A nonzero DRA thread request return code that causes the thread TCB to be collapsed.
- Termination using a IMS DB /STOP REGION command.

Failed thread TCBs are not automatically recreated. The thread TCB number increases again if a new thread is created to process a SCHED request. If the number of thread TCBs is above the MINTHRD value and all thread activity ceases normally, the number of thread TCBs left in the DRA will be the MINTHRDD value.

During CCTL processing, the number of active DRA threads occupying thread TCBs varies from 0 to the MAXTHRD number. Active DRA threads indicate that at least one SCHED request has been made but not any TERMTHRD requests. If the number of non-active thread TCBs becomes too large, the DRA automatically collapses some thread TCBs to release IMS DB resources.

The status of DRA thread TCBs can be evaluated from the output of the /DISPLAY CCTL ALL command, except for one case.

**Related Reading:** See *IMS Version 9: Command Reference* for examples of this command.

If there were no thread failures, the output might show fewer thread TCBs than the MINTHRD value because of internal short lived conditions. In fact, the actual number of thread TCBs **does** equal the MINTHRD.

## DRA Thread Statistics

DRA thread statistics are returned for a SCHED request and for any DRA requests that terminate a UOR. The statistics are in a CCTL area that is pointed to by the PAPLSTAT field. The PAPL listing maps this area, as shown in the following tables. The statistics also appear in the IMS DB log records X'08' (SCHED) and X'07' (UOR terminate).

Table 79. Information Provided for the Schedule Process:

PAPL Field	Field Length (Hexadecimal)	Contents
PAPLNPSB	8	PSB name
PAPLPOOL	8	Elapsed wait time for pool space (packed: microseconds)
PAPLINTC	8	Elapsed wait time - intent conflict (packed: microseconds)
PAPLSCHT	8	Elapsed time for schedule process (packed: microseconds)
PAPLTIMO	8	Elapsed time for DB I/O (packed: microseconds)
PAPLTLOC	8	Elapsed time for DI locking (packed: microseconds)
PAPLDBIO	4	Number of DB I/Os

Table 80. Information Provided at UOR Termination:

PAPL Field	Field Length (Hexadecimal)	Contents
PAPLGU1	4	Number of database GU calls issued
PAPLGN	4	Number of database GN calls issued

Table 80. Information Provided at UOR Termination: (continued)

PAPL Field	Field Length (Hexadecimal)	Contents
PAPLGNP	4	Number of database GNP calls issued
PAPLGHU	4	Number of database GHU calls issued
PAPLGHN	4	Number of database GHN calls issued
PAPLGHNP	4	Number of database GHNP calls issued
PAPLISRT	4	Number of database ISRT calls issued
PAPLDLET	4	Number of database DLET calls issued
PAPLREPL	4	Number of database REPL calls issued
PAPLTOTC	4	Total number of DL/I database calls
PAPLTENQ	4	Number of test enqueues
PAPLWTEQ	4	Number of WAITS on test enqueues
PAPLTSDQ	4	Number of test dequeues
PAPLUENQ	4	Number of update enqueues
PAPLWUEQ	4	Number of WAITS on updates and enqueues
PAPLUPDQ	4	Number of update dequeues
PAPLEXEQ	4	Number of exclusive enqueues
PAPLWEXQ	4	Number of WAITS on exclusive enqueues
PAPLEXDQ	4	Number of exclusive dequeues
PAPLDATS	8	STCK time schedule started
PAPLDATN	8	STCK time schedule completed
PAPLDECL	2	Number of DEDB calls
PAPLDERD	2	Number of DEDB read operations
PAPLMSCS	2	Reserved for Fast Path
PAPLOVFN	2	Number of overflow buffers used
PAPLUOWC	2	Number of UOW contentions
PAPLBFWT	2	Number of WAITS for DEDB buffers
PAPLUSSN	4	Unique schedule sequence number
PAPLCTM1	4	Elapsed UOR CPU time (for thread TCB) (For timer units, see z/OS STIMER macro)

## DRA Statistics

DRA statistics are contained in the returned PAPL as a result of a DRA TERM request, or in the Control exit routine's PAPL when it is called for DRA termination. This routine is called when the DRA fails or when a previous Control exit routine invocation resulted in return code 4.

The statistics in the returned PAPL are:

1. The number of times the MAXTHRD value was reached.
2. The number of times the MINTHRD value was reached (only includes the times the value is reached when the thread TCB number is decreasing.)
3. The largest number of thread TCBs ever reached during this DRA session. (This is the number of TCBs, not the number of DRA threads, so it is at least the minimum thread value.)

4. The time (in seconds) during which the DRA thread TCB count was at the MAXTHRD value.

You can find the field names for the previous statistics in the PAPL extensions for the TERM PAPL and control exit routine PAPL.

Before attempting to evaluate the statistics DRA performance, remember:

- If the DRA is using the maximum number of threads (MAXTHRD), when the DRA receives any new SCHED requests it will make these requests wait until a thread is available.
- As active threads become available (for example, as a result of TERMTHRD call), some of the available threads might be collapsed.

The above facts can adversely affect performance, but both improve IMS DB resource availability because fewer DRA threads require fewer IMS DB resources. The IMS DB resources (PSTs) are then available for other BMPs or other CCTLS to use.

Statistics 1, 2, and 4 can serve as measures of the two facts mentioned above, and will help you decide how to balance performance and resource usage. For the sake of the discussion here, these statistics are presented solely from a performance point of view (for example, assume only 1 CCTL connected to a IMS DB).

### Evaluating the DRA Statistics

If statistics 1 and 4 are high, a SCHED request had to wait for an available thread many times. To improve performance, raise the MAXTHRD value.

The impact of statistic 2 on performance can only be estimated if thread activity history is known (the DRA does not provide this history but the CCTL can). If activity is steady, little thread collapsing occurs and statistic 2 is meaningless. If activity fluctuates a lot, statistic 2 can be useful.

- If statistic 2 is 0, much thread collapsing might occurring, but the MINTHRD value was never reached.
- If statistic 2 is not zero, the MINTHRD value was reached and at those points, thread collapsing was stopped, thus enhancing performance. Therefore, if you have highly fluctuating thread activity, you can improve performance by raising MINTHRD until statistic 2 has a nonzero value.

Finally, statistic 3 can be useful for adjusting your MAXTHRD value.

**Note:** These statistics are useful in determining MINTHRD and MAXTHRD definitions. When MINTHRD=MAXTHRD, these statistics will be of no value.

## Tracing

There is no tracing (logging) of activity in the DRA, but there is tracing in IMS DB of DL/I and Fast Path activity. The setup and invocation of DL/I tracing for IMS DB is the same as for IMS. The output trace records for CCTL threads contain the recovery token.

Fast Path tracing in IMS DB is different from IMS. Fast Path tracing in IMS DB is activated when a SCHED request to the DRA has the PAPLFTRD equal to ON (Fast Path trace desired for this UOR).

When this UOR completes, a trace output file is closed and sent to SYSOUT Class A.



If a thread request fails during Fast Path processing, the DRA might return the PAPL with the PAPLFTRR field equal to ON. This recommends to the CCTL that it request the PAPLFTRD field be equal to ON (Fast Path trace desired) in the SCHED PAPL if this failing transaction is run again by the CCTL.

## Sending Commands to IMS DB

In an IMS DB warm standby or IMS/ESA® XRF environment, a CCTL might desire to have the IMS alternate system become the primary IMS system. To do this without operator intervention, a CCTL can use a z/OS SVC 34 to broadcast an emergency restart command to a IMS DB alternate, or a SWITCH command to an IMS XRF alternate. These are the only IMS commands that can be done using this interface. The command verb can be preceded by either the command recognition character or the 4-character IMS identification that is in the PAPLDBCT field of the INIT PAPL.

## Problem Determination

Failed DRA requests have a nonzero value in the PAPLRETC field of the PAPL returned to the CCTL. The format of PAPLRETC is:

HHSSUUU

Where: HH= X'00'- No output

**UUU** IMS DB return codes

X'88'- No output

**SSS** All z/OS non-retryable abend codes (for example, 222, 13E) or,

**UUU** IMS abend codes (775, 777, 844, 849, 2478, 2479, 3303)

X'84'- SNAP only

**UUU** IMS abend codes (260, 261, 263)

X'80'- SDUMP/SNAP provided

**SSS** All the z/OS retryable abend codes

**UUU** All IMS abend codes not listed above

Diagnostic information is provided by the DRA in the form of an SDUMP, or a SNAP dataset output. For X'80', the SDUMP is attempted first. If it fails, SNAP is done. For X'84', no SDUMP is attempted, but a SNAP is attempted.

A z/OS or IMS abend code failure results in DRA thread termination and thread TCB collapse. A IMS DB return code has no affect on the DRA itself or the thread TCB.

DRA thread TCB failures that occur when not processing a thread request result in a SDUMP/SNAP process. DRA control TCB failures that occur when not processing a DRA request result in a SDUMP/SNAP process and the Control exit routine is called. For a SCHED type of thread request, a failure with X'80' or X'84' can result in either SNAP or SDUMP.

### SDUMP

SDUMP output contains:

- The IMS control region.
- DLISAS address space.



- Key 0 and key 7 CSA.
- Selected parts of DRA private storage, including the ASCB, TCB, and RBs.

You can format the IMS control blocks by using the Offline Dump Formatter (ODF).

**Related Reading:** The ODF is described in *IMS Version 9: Diagnosis Guide and Reference*.

The ODF will not format DRA storage. You can use IPCS to format the z/OS blocks in the CCTL's private storage.

DRA SDUMPS have their own SDUMP options. As a result of this, any CHNGDUMP specifications **cannot** cause sections of DRA SDUMPS to be omitted. If these specifications aren't in the DRA's list of options, they can have an additive effect on DRA SDUMPS.

### **SNAPs**

The SNAP dump datasets are dynamically allocated whenever a SNAP dump is needed. A parameter in the DRA Startup Table defines the SYSOUT class.

The SNAP output contains:

- Selected parts of DRA private storage, including the ASCB, TCB, and RBs.
- IMS DB's control blocks.



---

## Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
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 is for planning purposes only. The information herein is subject to change before the products described become available.

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. 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 IBM's application programming interfaces.

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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Programming Interface Information

This book is intended to help the application programmer write IMS application programs. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IMS.

General-use programming interfaces allow the customer to write programs that obtain the services of IMS.

However, this book also documents Product-sensitive Programming Interface and Associated Guidance 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 IMS. 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 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 chapter or section or by the following marking:

Product-sensitive Programming Interface and Associated Guidance Information.

---

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

BookManager	Language Environment
C/370	Library Reader
C/MVS	MVS
CICS	MVS/ESA
CICS/ESA	OS/390
DB2	RACF
IBM	SAA
IMS	z/OS
IMS/ESA	

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States, other countries or both.

Other company, product, and service names may be trademarks or service marks of others.

---

## Product Names

In this book, the following licensed programs have shortened names:

- “C/C++ for MVS/ESA” is referred to as either “C/MVS” or “C++/MVS”.
- “COBOL for MVS & VM” is referred to as “COBOL”.
- “DB2 for MVS/ESA” is referred to as “DB2”.
- “Language Environment for MVS & VM” is referred to as “Language Environment”.
- “PL/I for MVS & VM” is referred to as “PL/I”.

## Bibliography

This bibliography includes all the publications cited in this book, including the publications in the IMS library.

- *CICS/ESA Application Programming Guide*, SC33-1169
- *CICS/ESA Application Programming Reference*, SC33-1170
- *CICS/ESA CICS-IMS Database Control Guide*, SC33-1184
- *CICS/MVS Installation Guide*, SC33-0506
- *CICS/ESA System Definition Guide*, SC33-1164
- *IBM Language Environment Installation and Customization on MVS*, SC26-4817
- *IBM Language Environment for MVS & VM Programming Guide*, SC26-4818
- *MVS/ESA: JCL Reference MVS/ESA System Product: JES2 Version 5*, GC28-1479
- *MVS/ESA System Programming Library: Application Development Guide*, GC28-1852
- *System Application Architecture Common Programming Interface: Resource Recovery Reference*, SC31-6821
- *IBM TSO Extensions for MVS/REXX Reference*, SC28-1883

### IMS Version 9 Library

ZES1-2330	ADB	<i>IMS Version 9: Administration Guide: Database Manager</i>
ZES1-2331	AS	<i>IMS Version 9: Administration Guide: System</i>
ZES1-2332	ATM	<i>IMS Version 9: Administration Guide: Transaction Manager</i>
ZES1-2333	APDB	<i>IMS Version 9: Application Programming: Database Manager</i>
ZES1-2334	APDG	<i>IMS Version 9: Application Programming: Design Guide</i>
ZES1-2335	APCICS	<i>IMS Version 9: Application Programming: EXEC DLI Commands for CICS and IMS</i>
ZES1-2336	APTM	<i>IMS Version 9: Application Programming: Transaction Manager</i>
ZES1-2337	BPE	<i>IMS Version 9: Base Primitive Environment Guide and Reference</i>
ZES1-2338	CR	<i>IMS Version 9: Command Reference</i>

ZES1-2339	CQS	<i>IMS Version 9: Common Queue Server Guide and Reference</i>
ZES1-2340	CSL	<i>IMS Version 9: Common Service Layer Guide and Reference</i>
ZES1-2341	CG	<i>IMS Version 9: Customization Guide</i>
ZES1-2342	DBRC	<i>IMS Version 9: DBRC Guide and Reference</i>
ZES1-2343	DGR	<i>IMS Version 9: Diagnosis Guide and Reference</i>
ZES1-2344	FAST	<i>IMS Version 9: Failure Analysis Structure Tables (FAST) for Dump Analysis</i>
ZES1-2346	OLR	<i>IMS Version 9: HALDB Online Reorganization Guide and Reference</i>
ZES1-2347	JGR	<i>IMS Version 9: IMS Java Guide and Reference</i>
ZES1-2348	IIV	<i>IMS Version 9: Installation Volume 1: Installation Verification</i>
ZES1-2349	ISDT	<i>IMS Version 9: Installation Volume 2: System Definition and Tailoring</i>
ZES1-2350	INTRO	<i>IMS Version 9: An Introduction to IMS</i>
ZES1-2351	MIG	<i>IMS Version 9: Master Index and Glossary</i>
ZES1-2352	MC1	<i>IMS Version 9: Messages and Codes, Volume 1</i>
ZES1-2353	MC2	<i>IMS Version 9: Messages and Codes, Volume 2</i>
ZES1-2354	OTMA	<i>IMS Version 9: Open Transaction Manager Access Guide and Reference</i>
ZES1-2355	OG	<i>IMS Version 9: Operations Guide</i>
GC17-7831	RPG	<i>IMS Version 9: Release Planning Guide</i>
ZES1-2358	URDBTM	<i>IMS Version 9: Utilities Reference: Database and Transaction Manager</i>
ZES1-2359	URS	<i>IMS Version 9: Utilities Reference: System</i>

#### Supplementary Publications

GC17-7825	LPS	<i>IMS Version 9: Licensed Program Specifications</i>
ZES1-2357	SOC	<i>IMS Version 9: Summary of Operator Commands</i>

**Publication Collections**

LK3T-7213	CD	IMS Version 9 Softcopy Library
LK3T-7144	CD	IMS Favorites
LBOF-7789	Hardcopy and CD	Licensed Bill of Forms (LBOF): IMS Version 9 Hardcopy and Softcopy Library
SBOF-7790	Hardcopy	Unlicensed Bill of Forms (SBOF): IMS Version 9 Unlicensed Hardcopy Library
SK2T-6700	CD	OS/390 Collection
SK3T-4270	CD	z/OS Software Products Collection
SK3T-4271	DVD	z/OS and Software Products DVD Collection

**Accessibility Titles Cited in this Book**

SA22-7787	z/OS V1R1.0 TSO Primer
SA22-7794	z/OS V1R1.0 TSO/E User's Guide
SC34-4822	z/OS V1R1.0 ISPF User's Guide, Volume 1



## Index

### Special characters

#### !token

IMSQUERY function 280  
STORAGE command 278

#### . (period) usage

null or void placeholder 271  
parsing, transparent additions 271  
REXX 269

\*mapname 274, 275

### Numerics

12-byte time stamp, field in I/O PCB 94

8-blanks (null) 154

## A

abend statement 311

#### accessing

GSAM databases 219

AD/Cycle C/370 117

addressability to UIB, establishing 103

addressing environments 262, 267

addressing mode (AMODE) 120

#### AIB (application interface block)

address return 114

AIB identifier (AIBID) 98

in APSB call 150  
in CHKP (basic) call 151  
in CHKP (symbolic) call 152  
in DPSB call 153  
in GMSG call 154  
in GSCD call 156  
in ICMD call 157  
in INIT call 159

AIB Identifier (AIBID)

in INQY call 164

AIBOLEN (maximum output area length)

in ICMD call 157

AIBERRXT (reason code) 99

AIBFUNC (subfunction code)

in DPSB call 153  
in GMSG call 154

#### AIBLEN

in GSCD call 156  
in INIT call 159

AIBLEN (DFSAIB allocated length)

in APSB call 150  
in CHKP (basic) call 151  
in CHKP (symbolic) call 152  
in DPSB call 153  
in GMSG call 154  
in ICMD call 157  
in INQY 164

AIBOLEN (maximum output area length) 99

in CHKP (symbolic) call 152  
in GMSG call 155

AIB (application interface block) *(continued)*

AIBOLEN (maximum output area length)  
*(continued)*

in GSCD call 157  
in INIT call 159  
in INQY call 164

AIBOAUSE (used output area length) 99

in GMSG call 155

AIBOLEN (maximum output area length)

in ICMD call 157

AIBREASN (reason code) 99

AIBRSA1 (resource address) 99

AIBRSNM1

in GSCD call 157  
in INIT call 159

AIBRSNM1 (resource name) 98

in APSB call 150  
in CHKP (basic) call 151  
in CHKP (symbolic) call 152  
in DPSB call 153  
in GMSG call 155  
in INQY call 164

AIBRSNM2

in APSB call 150  
in CHKP (basic) call 151

AIBSFUNC (subfunction code) 98

in INQY call 164

and program entry statement 114

description 111

DFSAIB allocated length (AIBLEN) 98

fields 98

interface, REXX 267

mask 98, 99

specifying 98

storage, defining 112

subfunction, setting 277

#### AIB mask

specifying 98

AIBERRXT (reason code) 99

AIBID (AIB identifier) field, AIB mask 98

AIBLEN (DFSAIB allocated length) field, AIB mask 98

AIBOLEN (maximum output area length) field, AIB mask 99

AIBOAUSE (used output area length) field, AIB mask 99

AIBREASN (reason code)

AIB mask, field 99

AIBREASN (reason code) field, AIB mask 99

AIBRSA1 (resource address) field, AIB mask 99

AIBRSNM1 (resource name) field, AIB mask 98

AIBSFUNC (subfunction code) field, AIB mask 98

AIBTDLI interface

See AIB (application interface block)

Allocate PSB (APSB) call 150

format 150

parameters 150

usage 150

AMODE 120

AND  
 dependent 212  
 independent 213  
 logical 199

AO (automated operator) application  
 GMSG call 154  
 ICMD call 157  
 RCMD call 172

AOI token  
 usage 155

APPC environment 262

application program  
 DB Batch environment, in 12  
 DBCTL environment, in 10  
 DBD, using 18  
 deadlock occurrence, in 163  
 environments 9  
 HALDB environment  
 scheduling 119  
 interface 9  
 PSB, using 18  
 sample hierarchies 20

applications, sample 45

APSB (Allocate PSB) call 150  
 format 150  
 parameters 150  
 usage 150

area  
 in CHKP (symbolic) call 152

area length  
 in CHKP (symbolic) call 152

areas  
 I/O 105

assembler language  
 DL/I call format, example 80  
 DL/I call-level sample 53  
 DL/I program structure 50  
 entry statement 112  
 parameters, DL/I call format 79  
 program entry 112  
 register 1, program entry 112  
 return statement 112  
 SSA definition examples 107  
 syntax diagram, DL/I call format 78  
 UIB, specifying 102

## B

backout point, intermediate  
 backing out 254

Basic Checkpoint (CHKP Basic) 150  
 format 151  
 parameters 151  
 usage 151

batch environments  
 DB  
 TM 9

batch programs  
 assembler language 50  
 C language 56  
 COBOL 60

batch programs (*continued*)  
 deadlock occurrence, in 163  
 maintaining integrity 253  
 overview 12  
 Pascal 68  
 PL/I 71  
 structure 12

batch regions and GSAM 227

BILLING segment 21

BMPs, transaction-oriented  
 ROLB 252

Boolean operators  
 dependent AND 212  
 independent AND 213  
 logical AND 199  
 logical OR 199  
 SSA, coding 107

## C

C language 113  
 \_\_pcblist 113  
 batch program, coding 56  
 DL/I call formats, example 83  
 DL/I program structure 56  
 entry statement 112, 113  
 exit 113  
 I/O area 83  
 parameters, DL/I call format 81  
 PCBs, passing 113  
 return statement 112  
 SSA definition examples 108  
 syntax diagram, DL/I call format 80  
 system function 113

C/C++ for MVS/ESA 117

call functions, DL/I 321

CALL statement 312  
 CALL DATA 315  
 CALL DATA statement internal field 315  
 CALL FUNCTION 312

call-level programs, CICS online 13

calls, DB  
 CIMS 121  
 CLSE 123  
 DEQ 123  
 DLET 125  
 FLD 126  
 GHNP 132  
 GHU 135  
 GN 128  
 GNP 132  
 GU 135  
 ISRT 138  
 OPEN 141  
 POS 142  
 REPL 145

calls, system service  
 APSB (allocate PSB) 150  
 CHKP (basic) 150  
 CHKP (symbolic) 151  
 GMSG (get message) 154

- calls, system service *(continued)*
  - ICMD (issue command) 157
  - INIT (initialize) 159
  - INQY (inquiry) 163
  - LOG (log) 169
  - PCB (schedule a PSB) 171
  - RCMD (retrieve command) 172
  - ROLB (roll back) 173
  - SETS/SETU (set a backout point) 176
  - SNAP 177
  - STAT (statistics) 180
  - SYNC (synchronization point) 182
  - TERM (terminate) 183
  - XRST (extended restart) 184
- calls, system service
  - SETS/SETU (set a backout point)
    - backing out to an intermediate backout point 254
    - using 254
- CCTL (coordinator controller)
  - design recommendation 375
  - in DBCTL environment 11
  - performance considerations
    - thread monitoring 376
- CEETDLI 117
  - address return 114
  - interface to IMS 118
  - program entry statement 77, 114
- checkpoint (CHKP)
  - call, necessary information 49
- checkpoint call
  - See CHKP call
- checkpoints (CHKP)
  - issuing 249
- CHKP (basic checkpoint) call
  - description 150
  - format 151
  - parameters 151
  - usage 151
- CHKP (checkpoint) call, necessary information 49
- CHKP (symbolic checkpoint) call
  - description 151
  - format 151
  - parameters 152
  - usage 153
  - with GSAM 224
- CHKP call function 318
- CHNG call function 318
- CICS DL/I call program, compiling 49
- CICS online programs 171
  - assembler language, sample 53
  - COBOL, establishing addressability 68
  - COBOL, optimization feature 68
  - COBOL, sample 63
  - PCB call 171
  - PL/I, sample 74
  - structure 13
  - TERM call 183
- CIMS call
  - description 121
  - format 121
  - parameters 122
- CIMS call *(continued)*
  - usage 122
- class, record segment 35
- closing a GSAM database explicitly 123, 222
- CLSE (Close) call
  - description 123
  - format 123
  - parameters 123
  - usage 123
- CMD call function 318
- COBOL
  - CICS online, establishing addressability 68
  - CICS online, optimization feature 68
  - DL/I call formats, example 85
  - DL/I call-level, sample 63
  - DL/I program structure 60
  - entry statement 112
  - parameters, DL/I call format 84
  - return statement 112
  - SSA definition examples 109
  - syntax diagram, DL/I call format 83
  - UIB, specifying 102
- COBOL for MVS & VM and Language Environment 117
- COBOL/370 and Language Environment 117
- codes, status
  - checking 17
  - logical relationships 217
- coding rules
  - SSA 106
- command codes 24, 30
- C
  - description 29
  - SSAs 24
- D
  - examples 27, 30
  - Get calls 30
  - ISRT call 31
  - P processing option 30
- DEDBs 28
  - description 27
- DL/I calls 28
- F
  - Get calls 31
  - HERE insert rule 140
  - ISRT call 32
  - restrictions 247
- L
  - FIRST insert rule 33, 140
  - Get calls 33
- M 40
- N 33
- Null 39
- overview 27
- P 34
- Q 34, 256
- qualified SSAs 27
- R 41
- reference 301
- restrictions 106
- S 42

command codes (*continued*)  
 subset pointers 28, 240, 242  
 summary 27  
 unqualified SSAs 27  
 V 38  
 Z 44

#### Command codes

U 37

#### COMMENT statement

conditional (T) 332  
 unconditional (U) 332

#### commit point processing

DEDB 245  
 MSDB 235

#### COMPARE statement

COMPARE AIB 335  
 COMPARE DATA 333  
 COMPARE PCB 335  
 introduction 333

#### concatenated datasets

GSAM 227

#### concatenated key and PCB mask 97, 221

#### concatenated segments, logical relationships 216

#### connector 128

#### CTL (PUNCH) statement 340

#### current position

determining 189  
 multiple positioning 203  
 qualification 37  
 unsuccessful calls 194

## D

#### data areas, coding 48

#### data availability, status codes 96

#### data entry database (DEDB)

*See* DEDB (data entry database)

#### data locking 236

#### data mapping, define with MAXDEF command 272

#### data redundancy, reducing 214

#### data structures 49

#### database

administrator 20

#### calls

Fast Path 247  
 summary 297

DB PCB, name 96, 221

deallocating resources 153

example, medical hierarchy 20  
 position

after XRST 186  
 determining 189  
 establishing using GU 137  
 multiple positioning 203  
 unsuccessful calls 194

recovery with ROLL call 252

recovery, back out changes 251

sample hierarchy 20

#### database management calls 16

#### database resource adapter 355

#### DB batch, program considerations 12

#### DB PCB

database name 96, 221

entry statement, pointer 220

fields 95, 96

in GSCD 156

key feedback area 221

key feedback area length field 97, 221

mask 95, 96

fields 221

fields, GSAM 220

name 220

relation 95

multiple DB PCBs 208

number of sensitive segments field 97

processing options field 97, 221

relation to DB PCB 95

secondary indexing, contents 214

segment level number field 96

segment name field 97

sensitive segments 97

status code field 96, 221

status codes

NA 160

NU 160

#### DB PCB (database program communication block)

#### masks

DB PCB 95

#### DB PCB mask

general description 95

specifying 95

#### DB PCB mask, GSAM reference 111

#### DB/DC environment

overview 10

#### DBA 20

#### DBB batch region 227

#### DBD (database description), description 18

#### DBDCTL environment

CCTL 11

overview 10

using DRA 10

using ODBA 11

#### DBQUERY

using with INIT call 160

#### deadlock occurrence

application programs 163

batch programs, in 163

#### debugging, IMSRXTRC 272

#### DEDB

multiple qualification statements 201

#### DEDB (data entry database)

call restrictions 246

command codes 39

DL/I calls 246

PCBs and DL/I calls 91

#### processing

commit point 245

data locking 236

fast path 229

H option 245

overview 231

P option 245

- DEDB (data entry database) *(continued)*
  - processing *(continued)*
    - POS call 242
    - subset pointers 238
  - root segments, order 132
  - updating with subset pointers 238
- DEDB(data entry database)
  - data locking 236
  - updating segments 231
- define a data mapping with MAXDEF command 272
- delete call
  - See DLET Call
- dependent AND 212
- dependents, direct 231
- DEQ (Dequeue) call
  - and Q command code 35, 124
  - description 123
  - format
    - Fast Path 124
    - full function 123
  - parameters
    - Fast Path 124
    - full function 124
  - restrictions 125
  - summary 297
  - usage 124
- DEQ call function 318
- DFSDDLTO (DL/I Test Program)
  - See DL/I Test Program (DFSDDLTO)
- DFSDDLTO internal control statements
  - AB0C1 statement (INTERNAL CALL STATEMENT) 309
  - WTSR statement (INTERNAL CALL STATEMENT) 309
- DFSIVA3 45
- DFSIVA6 45
- DFSPRP
  - macro keywords 362
- DFSPSP00 (DRA startup table) 362
- DFSREXXU (Example User Exit Routine)
  - sample 307
- DFSSAM01 (Loads the Database) 288
- DL/I call formats, example
  - assembler language 80
  - C language 83
  - COBOL 85
  - Pascal 88
  - PL/I 90
- DL/I call functions 318, 321
  - special DFSDDLTO
    - END 330
    - SKIP 330
    - STAK 330
    - START 330
  - supported
    - CHKP 318
    - CHNG 318
    - CMD 318
    - DEQ 318
    - DELET 318
    - FLD 318
- DL/I call functions *(continued)* supported *(continued)*
  - GCMD 318
  - GHN 318
  - GNHP 318
  - GHU 318
  - GMSG 318
  - GN 318
  - GNP 318
  - GU 319
  - ICMD 319
  - INIT 319
  - INQY 319
  - ISRT 319
  - LOG 319
  - POS 319
  - PURG 319
  - RCMD 319
  - REPL 319
  - ROLB 319
  - ROLL 319
  - ROLS 320
  - ROLX 320
  - SETO 320
  - SETS 320
  - SNAP 320
  - STAT 320
  - SYNC 320
  - XRST 320
- DL/I calls (general information)
  - qualifying your calls 24
    - command codes 27
    - concatenated key 29
    - field 24
    - segment type 24
  - relationships to PCBs
    - FF PCBs 91
  - REXXTDLI 266
  - SSAs 24
  - types 24
- DL/I calls, database management
  - CIMS 121
  - CLSE 123
  - DEQ 123
  - DLET 125, 126
  - FLD 126, 128
  - general description 16
  - GNHP call 135
  - GHU call 137
  - GN 128, 132
  - GNHP call 131
  - GNP 132, 135
  - GU 135, 138
  - ISRT 138, 141
  - OPEN 141
  - POS 142, 145
  - REPL 145, 147
  - summary 16, 297
- DL/I calls, general information
  - coding 48
  - getting started with 12

- DL/I calls, general information (*continued*)
    - using 16
  - DL/I calls, system service 149, 150
    - APSB 150
    - CHKP 150, 151, 153
    - DPSB 153
    - GMSG 154
    - GSCD 156, 157
    - INIT 159, 163
    - INQY 163
    - LOG 169, 171
    - PCB 171, 172
    - ROLB 173, 174, 252
    - ROLL 174, 251
    - ROLS 175, 176
    - SETS/SETU 176, 177
    - SNAP 177, 180
    - STAT 180, 182, 183, 184
    - summary 16, 298
    - SYNC 182, 183
    - XRST 184
  - DL/I language interfaces
    - overview 77
    - supported interfaces 77
  - DL/I options
    - logical relationships 214
    - secondary indexing 211
  - DL/I program structure 12
  - DL/I return codes (REXX) 267
  - DL/I Test Program (DFSDDLTO)
    - control statements 309, 346
    - execution in IMS regions 350, 351
    - explanation of return codes 351
    - hints on usage 351, 353
    - JCL requirements 346, 350
    - overview 309
    - restarting input stream 348
  - DL/I, CICS online
    - getting started with 13
  - DL/I, ODBA interface
    - getting started with 15
  - DLET (Delete) call
    - description 125
    - format 125
    - parameters 123, 125, 141
    - SSAs 126
    - usage 126
    - with MSDB, DEDB or VSO DEDB 231
  - DLET call function 318
  - DLI batch region and GSAM 227
  - DLIINFO
    - . (period) usage 271
    - REXX extended command 270, 271
  - DLITCBL 113
  - DLITPLI 114
  - DOCMD exec 289
  - DPSB call
    - description 153
    - format 153
    - parameters 153
    - usage 153
  - DRA (database resource adapter) 355
    - CCTL function requests 366
      - INIT 366
      - RESYNC 368
      - TERM 369
    - CCTL recovery process 375
    - communicating with DBCTL 10
    - DRA statistics 378
    - enabling
      - CCTL 363
      - ODBA 364
    - initializing
      - CCTL 364
      - ODBA 364
    - macro keywords 362
    - multithreading 356
    - PAPL 375
    - problem determination 380
    - processing
      - CCTL requests 365
      - ODBA calls 366
    - startup table 362
      - DFSPZPxx 362
    - sync-point processing 358
      - in-doubt state 361
      - protocol 359
    - termination 375
    - thread
      - ODBA 356
      - processing 355
      - structure 355
    - thread function requests 369
      - ABTTERM 373
      - COMTERM 373
      - IMS 371
      - PREP 372
      - SCHED 369
      - SYNTERM 372
      - TERMTHRD 374
    - thread statistics 377
    - tracing 379
- ## E
- E (COMPARE) statement 333
  - enabling
    - data availability status codes 96
  - END call function 330
  - entry and return conventions 112
  - environment (REXX)
    - address 262, 267
    - determining 270
    - extended 267
  - equal-to relational operator 25
  - error routines 18
    - I/O errors 18
    - programming errors 18
    - system errors 18
    - types of errors 18
  - ESAF (External Subsystem Attach Facility) 10

## examples

- Boolean operators 200
- D command code 27, 31
- DFSDDLTO statements
  - COMMENT 333
  - DATA/PCB COMPARE 337
  - DD 348
  - DL/I call functions 321
  - IGNORE 339
  - OPTION 340
  - PUNCH 342
  - STATUS 344
  - SYSIN, SYSIN2, and PREINIT 349
  - WTO 346
  - WTOR 346
- FLD/CHANGE 235
- FLD/VERIFY 235
- L command code 33
- medical database 20
- multiple qualification statements 200
- N command code 33
- Null command code 39
- P command code 34
- path call 27
- SSAs, secondary indexing 212
- U Command Code 37
- UIB, defining 103
- V command code 38
- exceptional conditions 18
- EXECIO
  - example 288
  - managing resources 262
- explicitly opening and closing a GSAM database 222
- extended commands
  - See REXXIMS commands
- extended environment
  - See environment (REXX)
- extended functions
  - See IMSQUERY extended function
- Extended Restart (XRST) 153
  - description 184
  - parameters 184
  - position in database 186
  - restarting your program 185
  - restrictions 187
  - starting your program normally 185
  - usage 185
- External Subsystem Attach facility (ESAF) 10

**F**

- F command code
  - restrictions 247
- Fast Path
  - database calls 229
  - databases, processing 229
  - DEDB (data entry database) 229
  - FSA 127
  - MSDB (main storage database) 229
  - processing MSDBs and VSO DEDBs 231
  - subset pointers, using with DEDBs 238

Fast Path (*continued*)

- types of databases 229
- field
  - changing contents 234
  - checking contents: FLD/VERIFY 232
- Field (FLD) call
  - See FLD (Field) call
- field name
  - FSA 127, 233
  - SSA, qualification statement 24
- field search argument
  - description 232
  - reference 127
- field value
  - FSA 233
  - SSA, qualification statement 24, 25
- fields in a DB PCB mask 96, 221
- file I/O
  - See EXECIO
- FIRST insert rule, L command code 33
- fixed-length records 223
- FLD (Field) call
  - description 126, 231
  - FLD/CHANGE 234
  - FLD/VERIFY 232
  - format 126
  - FSAs 127
  - parameters 126
  - summary 297
  - usage 127
- FLD call function 318
- free space, identifying 244
- FSA (field search argument)
  - connector 128
  - description 232
  - Field name 127
  - FSA status code 127
  - Op code 128
  - operand 128
  - reference 127
  - with DL/I calls 232
- FSA status code 127
- full-function database
  - PCBs and DL/I calls 91
  - segment release 36

**G**

- GB (end of database)
  - return status code 30
- GCMD call function 318
- GE (segment not found)
  - return status code 30
- Get calls
  - D command code 30
  - F command code 31
  - function 318
  - L command code 33
  - Null command code 39
  - P command code 34
  - Q command code 34



- Get calls (*continued*)
    - U Command Code 37
    - V command code 38
  - get hold next (GHN)
    - usage 131
  - Get Message (GMSG) call
    - See GMSG call 154
  - GHN (get hold next)
    - usage 131
  - GHNP
    - call 132
    - hold form 135
  - GHU (Get Hold Unique)
    - description 137
  - GMSG call
    - description 154, 156
    - format 154
    - parameters 154
    - restrictions 156
    - use 155
  - GN (Get Next) call
    - description 128
    - format 128
    - hold form (GHN) 131
    - parameters 129
    - SSAs 131
    - usages 130
  - GNP (Get Next in Parent) call
    - description 132
    - effect in parentage 134
    - format 133
    - hold form (GHNP) 135
    - parameters 133
    - SSAs 135
    - usages 133
      - linking with previous DL/I calls 134
      - processing with parentage 134
  - GPSB (generated program specification block),
    - format 115
  - greater-than relational operator 25
  - greater-than-or-equal-to relational operator 25
  - group name, field in I/O PCB 94
  - GSAM (generalized sequential access method)
    - accessing databases 219
    - call summary 224
    - CHKP 224
    - coding considerations 224
    - data areas 111
    - data set
      - attributes, specifying 227
      - characteristics, origin 225
      - concatenated 227
      - DD statement DISP parameter 226
      - extended checkpoint restart 226
    - database, explicitly opening and closing 222
    - DB PCB mask, fields 220
    - DB PCB masks 111
    - description 219
    - designing a program 219
    - DLI or DBB region types 227
    - fixed-length records 222
  - GSAM (generalized sequential access method) (*continued*)
    - I/O areas 223
    - PCBs and DL/I calls 91
    - record formats 222
    - records, retrieving and inserting 221
    - restrictions on CHKP and XRST 224
    - RSA 111, 221
    - status codes 223
    - undefined-length records 222
    - variable-length records 222
    - XRST 224
  - GSCD (Get System Contents Directory) call
    - description 156
    - format 156
    - parameters 156
    - usage 157
  - GU (Get Unique) call 135
    - description 135
    - format 135
    - hold form (GHU) 137
    - parameters 136
    - restrictions 138
    - SSAs 137
    - usage 136
- ## H
- H processing option 245
  - HALDB (High Availability Large Database)
    - HALDB
      - application programs, scheduling against 119
    - HALDB partitions
      - data availability 18
      - error settings 18
      - handling 18
    - initial load of 119
    - restrictions for loading logical child segments 18
    - scheduling 18
    - status codes 18
  - HDAM
    - multiple qualification statements 201
  - HDAM database, order of root segments 132
  - HERE insert rule
    - F command code 32
    - L command code 33
  - hierarchic sequence 130
  - hierarchical database example, medical 20
  - hierarchy
    - data structures 49
    - sample database 20
  - High Availability Large Database (HALDB) 18
  - HOUSHOLD segment 21
- ## I
- I/O area
    - C language 83
    - coding 106
    - for XRST 185
    - in CHKP (symbolic) call 152



- I/O area (*continued*)
  - in GMSG call 155
  - in GSCD call 157
  - in INIT call 159
  - in INQY call 164
- I/O Area
  - specifying 105
- I/O Area (input/output area) 105
- I/O area length
  - in CHKP (symbolic) call 152
- I/O area returned
  - keywords 143
  - map of 143
- I/O PCB
  - in GSCD 156
  - in INIT call 159
  - PCBs and DL/I calls 91
- I/O PCB mask
  - 12-byte time stamp 94
  - general description 92
  - group name field 94
  - input message sequence number 93
  - logical terminal name field 92
  - message output descriptor name 93
  - specifying 92
  - status code field 93
  - userid field 93
  - userid indicator field 95
- ICMD call
  - commands that can be issued 158
  - description 157, 159
  - format 157
  - parameters 157
  - restrictions 159
  - use 158
- IGNORE (N or .) statement 339
- ILLNESS segment 20
- IMSQUERY extended function
  - arguments 280
  - usage 280
- IMSRXTRC command 270, 272
- independent AND 213
- indexed field in SSAs 212
- indexing, secondary
  - DL/I Returns 214
  - effect on program 211
  - multiple qualification statements 212
  - SSAs 211
  - status codes 214
- infinite loop, stopping 266
- INIT
  - using with DBQUERY 160
- INIT (Initialize) call
  - automatic INIT DBQUERY 161
  - database availability, determining 160
  - description 159
  - enabling data availability, status codes 161
  - enabling deadlock occurrence, status codes 162
  - format 159
  - INIT STATUS GROUPA 161
  - INIT STATUS GROUPB 162
- INIT (Initialize) call (*continued*)
  - parameters 159
  - performance 161
  - performance considerations (IMS online) 161
  - restrictions 163
  - status codes 161
  - usage 159
- INIT call function 319
- input for a DL/I program 48
- input message sequence number, field in I/O PCB 93
- INQY (Inquiry) call
  - description 163
  - format 164
  - map of INQY subfunction to PCB type 169
  - parameters 164
  - querying
    - data availability 165
    - environment 166
    - PCB 168
    - program name 169
    - restriction 169
    - return and reason codes 169
    - usage 164
- INQY call
  - querying
    - LERUNOPT, using LERUNOPT subfunction 168
- INQY call function 319
- INQY DBQUERY 165
- INQY ENVIRON, data output 166
- INQY FIND 168
- INQY PROGRAM 169
- inserting
  - first occurrence of a segment 31
  - last occurrence 33
  - segments 139
- inserting a segment
  - as first occurrence 32
  - as last occurrence 33
  - GSAM records 221
  - in sequence 31
  - path of segments 31
  - root 139
  - rules to obey 139
  - specifying rules 140
- integrity
  - batch programs 253
  - maintaining, database 250
  - using ROLB 251
    - MPPs and transaction-oriented BMPs 252
  - using ROLL 251
  - using ROLS 253
- interfaces, DL/I 117
- interfaces, DL/I.
  - See DL/I interfaces
- intermediate backout point
  - backing out 254
- internal control statements, summary 309
- ISRT (Insert) call
  - D command code 31
  - description 138
  - F command code 32

ISRT (Insert) call (*continued*)  
 format 138  
 L command code 33  
 loading a database 33  
 parameters 138  
 RULES parameter 32  
 SSAs 140  
 with MSDB, DEDB or VSO DEDB 231  
 ISRT call function 319  
 Issue Command (ICMD) call  
 See ICMD call 157  
 IVP Sample Application 45  
 IVPREXX exec 293  
 IVPREXX sample application 265

## J

JCL (job control language), requirements 346, 350

## K

key feedback area  
 DB PCB, length field 97  
 length field in DB PCB 221  
 overview 97  
 keys  
 concatenated 29

## L

L (CALL) statement 312  
 LANG= Option on PSBGEN for PL/I Compatibility with  
 Language Environment 118  
 Language Environment  
 LANG = option for PL/I compatibility 118  
 Language Environment for MVS & VM 117  
 language interfaces, DL/I 117  
 language interfaces, DL/I.  
 See DL/I interfaces  
 length of key feedback area 221  
 less-than relational operator 25  
 less-than-or-equal-to relational operator 25  
 level number, field in DB PCB 96  
 limiting  
 number of full-function database calls 35  
 link-editing, reference 53, 73  
 locating dependents in DEDBs  
 last-inserted sequential dependent, POS call 243  
 POS call 243  
 specific sequential dependent, POS call 242  
 lock class and Q command code 35  
 lock management 257  
 LOG (Log) call  
 description 169  
 format 169  
 parameters 169  
 restrictions 171  
 usage 170  
 LOG call function 319  
 logical AND, Boolean operator 199  
 logical child 214

logical child segments  
 restrictions for HALDBs 18  
 logical OR, Boolean operator 199  
 logical parent 214  
 logical relationships  
 effect on programming 216  
 introduction 214  
 logical child 214  
 logical parent 214  
 physical parent 214  
 processing segments 214  
 programming, effect 214  
 status codes 217  
 logical structure 214  
 logical terminal name, field in I/O PCB 92

## M

M command code  
 examples 40  
 subset pointers, moving forward 40  
 main storage database (MSDB)  
 See MSDB (main storage database)  
 managing subset pointers in DEDBs with command  
 codes 230  
 MAP definition (MAPDEF) 270, 272  
 map name  
 See \*mapname  
 MAP reading (MAPGET) 270, 274  
 MAP writing (MAPPUT) 270, 275  
 mapping  
 MAPDEF 272  
 MAPGET 274  
 MAPPUT 275  
 mask  
 AIB 98  
 DB PCB 95  
 MAXQ and Q command code 35  
 medical database example 20  
 description 20  
 segments 20  
 message output descriptor name, field in I/O PCB 93  
 mixed-language programming 119  
 modifiable alternate PCBs 249  
 MPPs  
 ROLB 252  
 MSDB (main storage database)  
 call restrictions 237  
 commit point processing 235  
 nonrelated 230  
 PCBs and DL/I calls 91  
 processing 231  
 data locking 236  
 terminal related 230  
 types  
 description 230  
 nonrelated 23  
 related 22  
 updating segments 231  
 MSDB(main storage database)  
 data locking 236

MSDBs (main storage database)  
 processing commit points 235  
 multiple  
 DB PCBs 208  
 positioning 203  
 processing 203  
 qualification statements 199  
 qualification statements, DEDB 201  
 qualification statements, HDAM 201  
 qualification statements, PHDAM 201  
 multiple positioning  
 advantages of 206  
 effecting your program 206  
 resetting position 208  
 MVS environment 262  
 MYLTERM 237

**N**

N command code 33  
 NA 160  
 name field, segment 24  
 nonrelated (non-terminal-related) MSDB 230  
 not-equal-to relational operator 25  
 not-found status code  
 description 194  
 position after 194  
 NU 160  
 Null command code 39

**O**

O (OPTION) Statement 339  
 op code 128  
 OPEN (Open) call  
 description 141  
 format 141  
 usage 142  
 operand  
 FSA 128  
 operation parameter, SNAP external call 179  
 operator  
 FSA 233  
 SSA 24  
 operators  
 Boolean 199  
 relational 199  
 OPTION statement 339  
 options, processing; field in DB PCB 97, 221  
 OR, logical 199  
 OS/VS COBOL and Language Environment 117  
 overriding  
 FIRST insert rule 33  
 HERE insert rule 32, 33  
 insert rules 140

**P**

P command code 34  
 P processing option 30, 245

parameters  
 assembler language, DL/I call format 79  
 C language, DL/I call format 81  
 COBOL, DL/I call format 84  
 Pascal, DL/I call format 87  
 PL/I, DL/I call format 89  
 parentage, P command code 34  
 PART exec 286  
 PARTNAME exec 287  
 PARTNUM exec 287  
 parts of DL/I program 12  
 Pascal  
 batch program, coding 68  
 DL/I call formats, example 88  
 DL/I program structure 68  
 entry statement 112, 114  
 parameters, DL/I call format 87  
 PCBs, passing 114  
 SSA definition examples 109  
 syntax diagram, DL/I call format 86  
 path call 30  
 D command code 30  
 definition 27  
 example 27  
 overview 27  
 PATIENT segment 20  
 PAYMENT segment 21  
 PCB (program communication block)  
 address list, accessing 102  
 DL/I calls, relationship 91  
 DLIINFO call 271  
 masks  
 description 13  
 GSAM databases 219  
 I/O PCB 92  
 modifiable alternate PCBs 249  
 types 115  
 PCB (schedule a PSB) call  
 description 171  
 format 171  
 parameters 171  
 usage 171  
 PCBINFO exec 284  
 PCHSEGTS 143  
 PCLBTSGTS 143  
 PCSEGRTS 143  
 period usage  
 See usage  
 PHDAM  
 multiple qualification statements 201  
 PHDAM database 132  
 physical parent 214  
 PL/I  
 batch program, coding 71  
 DL/I call formats, example 90  
 DL/I call-level sample 74  
 DL/I program, multitasking restriction 71  
 entry statement 112  
 parameters, DL/I call format 89  
 PCBs, passing 114  
 pointers in entry statement 114

PL/I (*continued*)  
 return statement 112  
 SSA definition examples 110  
 syntax diagram, DL/I call format 88  
 UIB, specifying 102  
 PL/I for MVS & VM and Language Environment 117  
 PLI/370 and Language Environment 117  
 PLITDLI 255  
 POS (Position) call  
 description 142, 242  
 examples 145  
 format 142  
 I/O area 143  
 parameters 142  
 unqualified  
 keywords 143  
 usage 145  
 POS call function 319  
 POS(positioning)=MULT(multiple) 203  
 position  
 establishing in database 137  
 positioning  
 after DLET 191  
 after ISRT 193  
 after REPL 193  
 after retrieval calls 190  
 after unsuccessful calls 194  
 after unsuccessful DLET or REPL call 194  
 after unsuccessful retrieval or ISRT call 195  
 CHKP, effect 249  
 current, after unsuccessful calls 194  
 determining 189  
 multiple 203  
 understanding current 189  
 PREINIT parameter, input restart 346  
 preloaded programs 120  
 processing  
 commit-point in DEDB 245  
 commit-point in MSDB 235  
 database, several views 208  
 DEDBs 238  
 Fast Path databases 229  
 GSAM databases 219  
 MSDBs and VSO DEBDs 231  
 multiple 203  
 options  
 field in DB PCB 97, 221  
 H (position), for Fast Path 245  
 P (path) 30  
 P (position), for Fast Path 245  
 segments in logical relationships 214  
 program  
 batch structure 12  
 design 48  
 restarting 249  
 program communication block  
 See PCB (program communication block)  
 program communication block.  
 See DB (database program communication block)  
 program deadlock 162

programming  
 guidelines 47  
 mixed language 119  
 secondary indexing 211  
 PSB (program specification block)  
 description 18  
 format 115  
 PSSEGHWM 143  
 PUNCH statement 340  
 PURG call function 319

## Q

Q command code 256  
 and the DEQ call 36  
 example 35  
 full function and segment release 36  
 lock class 35  
 MAXQ 35  
 qualification statement  
 coding 106  
 field name 24  
 field value 24, 25  
 multiple qualification statements 199  
 multiple qualification statements, DEDB 201  
 multiple qualification statements, HDAM 201  
 multiple qualification statements, PHDAM 201  
 overview 24  
 relational operator 24, 25  
 segment name 24  
 structure 24  
 qualified call  
 definition 24  
 overview 19, 24  
 qualified SSA  
 qualification statement 24  
 structure 24  
 structure with command code 27  
 qualifying  
 DL/I calls with command codes 27  
 SSAs 24

## R

R command code 41  
 RACF signon security 94  
 RACROUTE SAF 94  
 randomizing routine  
 exit routine 132, 201  
 RCMD call  
 description 172, 173  
 format 172  
 parameters 172  
 restrictions 173  
 use 173  
 reading segments in MSDBs 232, 237  
 record search argument  
 See RSA (record search argument)  
 regions, batch  
 DBB 227  
 DLI 227

- related (terminal related) MSDB 230
- relational operators
  - Boolean operators 199
  - independent AND 199
  - list 25
  - logical AND 199
  - logical OR 199
  - overview 25
  - SSA, coding 106
  - SSA, qualification statement 24
- REPL (Replace) call
  - description 145
  - format 145
  - N command code 33
  - parameters 146
  - SSAs 146
  - usage 146
    - with MSDB, DEDB or VSO DEDB 231
- REPL call function 319
- requesting a segment
  - using GU 136
- reserving
  - place for command codes 247
  - segment
    - command code 256
    - lock management 257
- resetting a subpointer 42
- residency mode (RMODE) 120
- Restart, Extended
  - parameters 184
  - position in database 186
  - restarting your program 185
  - restrictions 187
  - starting your program normally 185
  - usage 185
- Restart, Extended (XRST) 153
  - description 184
- restarting your program
  - XRST call 185
- restarting your program, basic checkpoints 249
- restrictions
  - CHKP and XRST with GSAM 224
  - database calls
    - to DEDBs 246
    - to MSDBs 237
  - F command code 32
  - number of database calls and Fast Path 35
- retrieval calls
  - D command code 30
  - F command code 31
  - L command code 33
  - status codes, exceptional 18
- Retrieve Command (RCMD) call
  - See RCMD call 172
- retrieving
  - dependents sequentially 132
  - first occurrence of a segment 31
  - last occurrence 33
  - segments
    - Q command code, Fast Path 35
    - Q command code, full function 35
  - retrieving (*continued*)
    - segments (*continued*)
      - sequentially 30
      - segments with D 30
- return codes
  - UIB 102, 303
- REXX
  - . (period) usage 269
  - calls
    - return codes 267
    - summary 267
    - syntax 267
  - commands
    - DL/I calls 266
    - summary 266
  - DL/I calls, example 269
  - execs
    - DFSSAM01 288
    - DOCMD 289
    - IVPREXX 293
    - PART 286
    - PARTNAME 287
    - PARTNUM 287
    - PCBINFO 284
    - SAY 283
    - IMSRXTRC, trace output 272
- REXX, IMS adapter
  - . (period) usage 271
  - address environment 262
  - AIB, specifying 268
  - description 261
  - DFSREXX0 program 261, 265
  - DFSREXX1 261
  - DFSREXXU user exit 261
  - DFSRR00 265
  - diagram 264
  - DL/I parameters 268
  - environment 270
  - example execs 283
  - feedback processing 268
  - I/O area 268
  - installation 261
  - IVPREXX exec 265
  - IVPREXX PSB 262
  - IVPREXX setup 262
  - LLZZ processing 268
  - LNKED requirements 261
  - non-TSO/E 261
  - PCB, specifying 268
  - programs 261
  - PSB requirements 261
  - sample generation 262
  - sample JCL 262
  - SPA processing 268
  - SRRBACK 261
  - SRRRCMIT 261
  - SSA, specifying 268
  - SYSEXEC DD 261, 262
  - system environment 261, 262
  - SYSTSIN DD 262
  - SYSTSPRT DD 261, 262

REXX, IMS adapter (*continued*)  
 TSO environment 261  
 TSO/E restrictions 261  
 ZZ processing 268

REXXIMS commands 272, 274  
*See also* IMSQUERY extended function  
 DLIINFO 270, 271  
 IMSRXTRC 270, 272  
 MAPDEF 270  
 MAPGET 270  
 MAPPUT 270, 275  
 SET 270, 276  
 SRRBACK 270, 277  
 SRRCMIT 270, 277  
 STORAGE 270, 278  
 WTL 270, 279  
 WTO 270, 279  
 WTOR 270, 279  
 WTP 270, 279

REXXTDLI commands 266

RMODE 120

ROLB  
 in MPPs and transaction-oriented BMPs 252

ROLB (Roll Back) call  
 compared to ROLL call 250  
 description 173, 252  
 format 173  
 maintaining database integrity 250  
 parameters 173  
 usage 252

ROLB call function 319

ROLL (Roll) call  
 compared to ROLB call 250  
 description 174, 251  
 format 174  
 maintaining database integrity 250

ROLL call function 319

ROLS  
 backing out to an intermediate backout point 254

ROLS (Roll Back to SETS) call  
 description 175  
 format 175  
 maintaining database integrity 250  
 parameters 175  
 TOKEN 253

ROLS call function 320

ROLX call function 320

routines, error 18

RSA (record search argument)  
 description 221  
 GSAM, reference 111  
 overview 221

rules  
 coding an SSA 106

RULES parameter  
 FIRST, L command code 33  
 HERE  
 F command code 32  
 L command code 33

## S

S (STATUS) statement 342

S command code  
 examples 42  
 subpointer, resetting 42

sample JCL 346

sample programs  
 call-level assembler language, CICS online 53  
 call-level COBOL, CICS online 63  
 call-level PL/I, CICS online 74

SAY exec 283

scheduling HALDBs 18  
 application programs, against 119

secondary indexes  
 multiple qualification statements 212

secondary indexing  
 DB PCB contents 214  
 effect on programming 211  
 information returned by DL/I 214  
 SSAs 211  
 status codes 214

secondary processing sequence 212

segment  
 requesting using GU 136

segment level number field 96

segment name  
 DB PCB, field 97  
 SSA, qualification statement 24

segment search argument  
*See* SSA (segment search argument)

segment search argument (SSA)  
 coding rules 106

segment, information needed 49

segments  
 in medical database example 20

sensitive segments in DB PCB 97

sequence  
 hierarchy 130

sequence field  
 virtual logical child, in 25

sequence, indication for statements 346

sequential dependent segments  
 how stored 231

sequential dependents 231  
 overview 231

SET command (REXX) 270, 276, 277

SET SUBFUNC command (REXX) 277

SET ZZ 277

SETO call function 320

SETO, DFSDDLTO  
 description 312

SETS  
 backing out to an intermediate backout point 254

SETS (Set a Backout Point) call  
 description 176, 254  
 format 176  
 parameters 176

SETS call function 320

setting  
 parentage with the P command code 34  
 subset pointer to zero 44



- SETU
  - backing out to an intermediate backout point 254
- SETU (Set a Backout Point Unconditional) call
  - description 176, 254
  - format 176
  - parameters 176
- SETU, call function 254
- signon security, RACF 94
- single positioning 203
- skeleton programs
  - assembler language 50
  - C language 56
  - COBOL 60
  - Pascal 68
  - PL/I 71
- SKIP call function 330
- SNAP call
  - description 177
  - format 177
  - parameters 177
  - status codes 180
- SNAP call function 320
- specifying
  - command codes for DEDBs 240
  - DB PCB mask 95
  - GSAM data set attributes 227
  - processing options for DEDBs 245
- Spool API
  - STORAGE command example 279
- SRRBACK command (REXX)
  - description 270
  - format, usage 277
- SRRCMIT command (REXX)
  - description 270
  - format, usage 277
- SSA (segment search argument)
  - coding
    - formats 107
    - restrictions 107
    - rules 106
  - coding rules 106
  - command codes 27
  - definition 24
  - overview 24
  - qualification statement 106
  - qualified 24
  - reference 106
  - relational operators 25
  - restrictions 106
  - segment name field 24, 106
  - structure with command code 27
  - unqualified 24
  - usage 126
    - command codes 27
    - DLET 126
    - GN 131
    - GNP 135
    - GU 137
    - guidelines 26
    - ISRT 140
    - multiple qualification statements 199
- SSA (segment search argument) *(continued)*
  - usage *(continued)*
    - REPL 146
    - secondary indexing 211
    - virtual logical child, in 25
- SSAs (segment search arguments)
  - unqualified 24
- STAK call function 330
- START call function 330
- STAT (Statistics) call
  - description 180
  - format 180
  - parameters 181
  - usage 182
- STAT call function 320
- status code
  - GE (segment not found) 30
- status codes
  - blank 17
  - checking 17
  - DB PCB, for 160
  - error routines 18
  - exception conditions 18
  - field in DB PCB 96, 221
  - FSA 233
  - GB, end of database 30
  - GSAM 223
  - H processing option 245
  - HALDB partitions 18
  - logical relationships 217
  - P processing option 245
  - retrieval calls 18
  - subset pointers 242
- status codes, field in I/O PCB 93
- STATUS statement 342
- storage
  - !token 278
  - STORAGE command 278
- STORAGE command (REXX)
  - description 270
  - format, usage 278
- subset pointer command codes
  - restrictions 28
- subset pointers
  - DEDB, managed by command codes 28
  - defining, DBD 240
  - defining, PCB 240
  - description 238
  - M command 40
  - preparing to use 240
  - R command code 41
  - resetting 42
  - S command code 42
  - sample application 39, 241
  - status codes 242
  - using 238
  - Z command code 44
- Summary
  - database management call 297
  - system service calls 298

summary of changes for DFSDDL0 internal control statements 309

summary of command codes 27

Symbolic Checkpoint (CHKP Symbolic) 151

- format 151
- parameters 152
- restrictions 153
- usage 153

SYNC (Synchronization Point) call

- description 182
- format 182
- parameters 183
- usage 183

SYNC call function 320

syntax diagram

- assembler language, DL/I call format 78
- C language, DL/I call format 80
- COBOL, DL/I call format 83
- Pascal, DL/I call format 86
- PL/I, DL/I call format 88

SYSIN input 346

SYSIN2 input processing 346

system service calls

*See also* DL/I calls, system service

- APSB (Allocate PSB) 150
- CHKP (Basic) 150
- CHKP (Symbolic) 151
- DPSB (deallocate) 153
- GMSG (Get Message) 154
- ICMD (Issue Command) 157
- INIT (Initialize) 159
- INQY (Inquiry) 163
- LOG (Log) 169
- PCB (schedule a PSB) 171
- RCMD (Retrieve Command) 172
- ROLB (Roll Back) 173
- SETS/SETU (Set a Backout Point) 176
- SNAP 177
- STAT (Statistics) 180
- SYNC (Synchronization Point) 182
- TERM (Terminate) 183
- XRST (Extended Restart) 184

## T

T (Comment) statement 332

TERM (Terminate) call

- description 183
- format 183
- usage 183

test program

*See* DL/I Test Program (DFSDDL0)

testing status codes 17

transaction-oriented BMPs

- ROLB 252

TREATMNT segment 21

TSO/E REXX

*See* REXX, IMS adapter

## U

U (Comment) statement 332

U Command Code 37

UIB (user interface block)

- defining, in program 102
- field names 104
- PCB address list, accessing 102
- return codes, accessing 102
- return codes, list 303

UIBDLTR

- introduction 103
- return codes, checking 303

UIBFCTR

- introduction 103
- return codes, checking 303

UIBPCBAL

- introduction 103
- return codes, checking 303

undefined-length records 221

unqualified call

- overview 19

unqualified calls, definition of 24

unqualified POS call

- I/O returned area

  - key words 143
  - map of 143

- keywords 143

unqualified SSA

- segment name field 24
- structure with command code 27
- usage with command codes 27

UOW boundary, processing DEDB 245

updating

- segments in an MSDB, DEDB or VSO DEDB 231

user interface block

*See* UIB (user interface block)

userid indicator, field in I/O PCB 95

userid, field in I/O PCB 93

## V

V command code 38

V5SEGRBA 143

variable-length records 221

virtual logical child 25

virtual storage option data entry database (VSO DEDB)

*See* VSO DEDB (virtual storage option data entry database), processing

VS COBOL II and Language Environment 117

VSAM, STAT call 182

VSO DEDB (virtual storage option data entry database), processing 231

## W

WAITAOI 154



WTL command (REXX)  
  description 270  
  format, usage 279  
WTO command (REXX)  
  description 270  
  format, usage 279  
WTO statement 345  
WTOR command (REXX)  
  description 270  
  format, usage 279  
WTOR statement 346  
WTP command (REXX)  
  description 270  
  format, usage 279

## **X**

XRST (Extended Restart) 153  
XRST (Extended Restart) call  
  description 184  
  format 184  
  parameters 184  
  restrictions 187  
  usage 185  
  with GSAM 224  
XRST call function 320

## **Z**

Z command code  
  examples 44  
  setting a subpointer to zero 44







Program Number: 5655-J38

IBM Confidential  
Printed in USA

ZES1-2333-00



Spine information:



IMS

Application Programming: Database Manager

Version 9