Enterprise PL/I for z/OS

Programming Guide

Version 5 Release 1

GI13-4536-00

Enterprise PL/I for z/OS

Programming Guide

Version 5 Release 1

GI13-4536-00

Note
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 525.

Second Edition (April 2017)

This edition applies to Version 5 Release 1 of Enterprise PL/I for z/OS and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

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

IBM Corporation, Department H150/090
555 Bailey Ave
San Jose, CA, 95141-1099
United States of America

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corporation 1999, 2017.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables xi

Figures xiii

Introduction xv

About this document xv
Runtime environment for Enterprise PL/I for z/OS xv
Using your documentation xv
Notation conventions used in this document . . . xvi
Conventions used xvi
How to read the syntax notation xvii
How to read the notational symbols xix
Summary of changes xx
Enhancements in this release xx
Enhancements from V4R5 xxiii
Enhancements from V4R4 xxiv
Enhancements from V4R3 xxv
Enhancements from V4R2 xxvi
Enhancements from V4R1. xxvii
Enhancements from V3R9 xxviii
Enhancements from V3R8 xxx
Enhancements from V3R7 xxxi
Enhancements from V3R6. xxxii
Enhancements from V3R5 xxxiii
Enhancements from V3R4 xxxiv
Enhancements from V3R3 xxxvi
Enhancements from V3R2 xxxvii
Enhancements from V3R1 xxxviii
Enhancements from VisualAge PL/I xxxix
How to send your comments xl
Accessibility xl

Part 1. Compiling your program . . . 1

Chapter 1. Using compiler options and facilities. 3

Compile-time option descriptions 3
AGGREGATE 7
ARCH 8
ASSERT 9
ATTRIBUTES 9
BACKREG. 10
BIFPREC 10
BLANK. 11
BLKOFF 12
BRACKETS 12
CASE 12
CASERULES 13
CEESTART 13
CHECK 14
CMPAT. 15
CODEPAGE 16
COMMON 16
COMPILE 17
COPYRIGHT 17
CSECT 18
CSECTCUT 18
CURRENCY 19
DBCS 19
DD 19
DDSQL. 20
DECIMAL. 20
DECOMP 22
DEFAULT 22
DEPRECATE 31
DEPRECATENEXT 32
DISPLAY 33
DLLINIT 33
EXIT. 33
EXPORTALL 34
EXTRN 34
FILEREF 34
FLAG 34
FLOAT 35
FLOATINMATH. 36
GOFF 37
GONUMBER 38
GRAPHIC 38
HEADER 39
IGNORE 39
INCAFTER 39
INCDIR 40
INCLUDE 40
INCPDS 41
INITAUTO 41
INITBASED 42
INITCTL 42
INITSTATIC 43
INSOURCE 43
INTERRUPT 44
JSON 44
LANGLVL. 45
LIMITS 45
LINECOUNT 47
LINEDIR 47
LIST 47
LISTVIEW 47
LP 49
MACRO 49
MAP 50
MARGINI 50
MARGINS. 50
MAXBRANCH 51
MAXGEN 52
MAXMEM. 52
MAXMSG 53
MAXNEST 53
MAXSTMT 54
MAXTEMP 54
MDECK 54

© Copyright IBM Corp. 1999, 2017

MSGSUMMARY 55
NAME 55
NAMES 56
NATLANG 56
NEST 56
NOT. 56
NULLDATE 57
NUMBER 57
OBJECT 58
OFFSET 58
OFFSETSIZE 58
ONSNAP 59
OPTIMIZE. 59
OPTIONS 60
OR 61
PP 61
PPCICS. 62
PPINCLUDE 63
PPLIST 63
PPMACRO 64
PPSQL 64
PPTRACE 65
PRECTYPE 65
PREFIX. 65
PROCEED. 66
PROCESS 66
QUOTE. 67
REDUCE 67
RENT 68
RESEXP 69
RESPECT 70
RTCHECK. 70
RULES 70
SEMANTIC 83
SERVICE 83
SOURCE 84
SPILL 84
STATIC. 84
STDSYS 84
STMT 85
STORAGE. 85
STRINGOFGRAPHIC 86
SYNTAX 86
SYSPARM 87
SYSTEM 87
TERMINAL 88
TEST 88
UNROLL 91
USAGE. 92
WIDECHAR 93
WINDOW. 93
WRITABLE 93
XINFO 95
XML. 97
XREF 97
Blanks, comments and strings in options 98
Changing the default options 99
Specifying options in the %PROCESS or *PROCESS statements. 99
Using % statements 100
Using the %INCLUDE statement 101
Using the compiler listing 102
Heading information 102
Options used for compilation 103
Preprocessor input 103
SOURCE program. 103
Statement nesting level 104
ATTRIBUTE and cross-reference table 104
Aggregate length table 105
Statement offset addresses 105
Storage offset listing 108
Expressions and attributes listing. 109
File reference table 109
Messages and return codes 110
Example 111

Chapter 2. PL/I preprocessors 117
Include preprocessor 117
Include preprocessor options in the configuration file 118
Include preprocessor options environment variable 118
Macro preprocessor 118
Macro preprocessor options 119
Macro preprocessor options in the configuration file 121
Macro preprocessor options environment variables 122
Macro preprocessor example 122
SQL preprocessor 123
Programming and compilation considerations 123
SQL preprocessor options 125
Coding SQL statements in PL/I applications 129
Manipulating LOB data 140
Suppressing SQL preprocessor messages . . . 144
CICS preprocessor. 145
Programming and compilation considerations 145
CICS preprocessor options 146
Coding CICS statements in PL/I applications 146
Writing CICS transactions in PL/I 147
Error-handling 147

Chapter 3. Using PL/I cataloged procedures 149
IBM-supplied cataloged procedures 149
Compile only (IBMZC) 150
Compile and bind (IBMZCB) 151
Compile, bind, and run (IBMZCBG). 153
Invoking a cataloged procedure 155
Specifying multiple cataloged procedure invocations 155
Modifying the PL/I cataloged procedures 156
EXEC statement 156
DD statement 157

Chapter 4. Compiling your program 159
Invoking the compiler under z/OS UNIX 159
Input files 159
Specifying compile-time options under z/OS UNIX 160
-qoption_keyword. 160

Single and multiletter flags 161
Invoking the compiler under z/OS using JCL . . 161
EXEC statement 162
DD statements for the standard data sets . . . 162
Listing (SYSPRINT) 164
Source Statement Library (SYSLIB) 164
Specifying options 165
Specifying options in the EXEC statement . . . 165
Specifying options in the EXEC statement using an options file 166

Chapter 5. Link-editing and running for 31-bit programs 167
Link-edit considerations for 31-bit programs . . . 167
Using the binder in 31-bit programs 167
Using the ENTRY card 167
Runtime considerations for 31-bit programs . . . 167
Formatting conventions for PRINT files . . . 168
Changing the format on PRINT files for 31-bit programs 168
Automatic prompting 169
Overriding automatic prompting 170
Punctuating long input lines 170
Punctuating GET LIST and GET DATA statements 171
Automatic padding for GET EDIT 171
Use of SKIP for terminal input 171
ENDFILE 171
SYSPRINT considerations for 31-bit programs . . 172
Using MSGFILE(SYSPRINT) 173
Using FETCH in your routines in 31-bit applications 173
Fetching Enterprise PL/I routines in 31-bit applications 174
Fetching PL/I MAIN routines in 31-bit applications 182
Fetching z/OS C routines in 31-bit applications 183
Fetching assembler routines in 31-bit applications 183
Invoking MAIN under TSO/E 183
Invoking MAIN under z/OS UNIX 184

Chapter 6. Link-editing and running for 64-bit programs 187
Link-edit considerations for 64-bit programs . . . 187
Using the binder in 64-bit programs 187
Using the ENTRY card in 64-bit programs . . . 187
Runtime considerations for 64-bit programs . . . 187
Formatting conventions for PRINT files for 64-bit programs 188
Automatic prompting 188
Overriding automatic prompting 188
Punctuating long input lines 189
Punctuating GET LIST and GET DATA statements 189
Automatic padding for GET EDIT 189
Use of SKIP for terminal input 190
ENDFILE 190
SYSPRINT considerations for 64-bit programs . . 190
Using FETCH in your routines in 64-bit applications 190
Fetching Enterprise PL/I routines in 64-bit applications 190
Fetching PL/I MAIN routines in 64-bit applications 191
Fetching assembler routines in 64-bit applications 191
Invoking MAIN under TSO/E 191
Invoking MAIN under z/OS UNIX 192

Chapter 7. Considerations for developing 64-bit applications 195
Using compiler options to build 64-bit applications 195
Using attributes HANDLE and POINTER under LP(64) 196
HANDLE attribute 196
POINTER attribute 196
Using ENTRY variables under LP(64) 197
Using built-in functions under LP(64) 197
Considerations for SQL programs 198
Communicating with 31-bit routines 200

Part 2. Using I/O facilities 203

Chapter 8. Using data sets and files 205
Allocating files 205
Associating data sets with files under z/OS . . . 207
Associating several files with one data set . . . 209
Associating several files with one data set . . . 209
Associating several data sets with one file . . . 209
Concatenating several data sets 210
Accessing HFS files under z/OS 210
Associating data sets with files under z/OS UNIX 211
Using environment variables 212
Using the TITLE option of the OPEN statement 213
Attempting to use files not associated with data sets 214
How PL/I finds data sets 214
Specifying characteristics using DD_DDNAME environment variables 214
Establishing data set characteristics 220
Blocks and records 221
Information interchange codes 221
Record formats 221
Data set organization 223
Labels 224
Data Definition (DD) statement 224
Using the TITLE option of the OPEN statement 226
Associating PL/I files with data sets 226
Specifying characteristics in the ENVIRONMENT attribute 228

Chapter 9. Using libraries 239
Types of libraries 239
Using a library 239
Creating a library 240
SPACE parameter 240
Creating and updating a library member 241

Example: Creating new libraries for compiled object modules 241
Example: Placing a load module in an existing library 242
Example: Updating a library member 242
Extracting information from a library directory . . 243

Chapter 10. Defining and using consecutive data sets 245
Using stream-oriented data transmission 245
Defining files using stream I/O 245
Defining stream files using PL/I dynamic allocation 246
Specifying ENVIRONMENT options 246
Creating a data set with stream I/O 249
Accessing a data set with stream I/O 253
Using PRINT files with stream I/O 254
Using SYSIN and SYSPRINT files for 31-bit programs 259
Using SYSIN and SYSPRINT files for 64-bit programs 259
Controlling input from the terminal 259
Format of data 261
Stream and record files 261
Defining QSAM files using PL/I dynamic allocation 262
Capital and lowercase letters 262
End-of-file 262
COPY option of GET statement 263

Chapter 11. Controlling output to the terminal 265
Format of PRINT files 265
Stream and record files 265
Output from the PUT EDIT command 266

Chapter 12. Using record-oriented data transmission 267
Specifying record format 268
Defining files using record I/O 268
Specifying ENVIRONMENT options 268
CONSECUTIVE 269
ORGANIZATION(CONSECUTIVE) 269
CTLASA | CTL360 270
LEAVE | REREAD 271
Creating a data set with record I/O 272
Essential information 272
Accessing and updating a data set with record I/O 273
Essential information 274
Example of consecutive data sets 274

Chapter 13. Defining and using regional data sets 279
Defining REGIONAL(1) data sets using PL/I dynamic allocation 281
Defining files for a regional data set 281
Specifying ENVIRONMENT options 281
Using keys with REGIONAL data sets 282
Using REGIONAL(1) data sets 282
Dummy Records 283
Creating a REGIONAL(1) data set 283
Accessing and updating a REGIONAL(1) data set 284
Essential information for creating and accessing regional data sets 287

Chapter 14. Defining and using VSAM data sets 291
Defining VSAM file using PL/I dynamic allocation 291
Using VSAM data sets 291
Running a program with VSAM data sets . . . 291
Pairing an alternate index path with a file . . . 292
VSAM organization 292
Keys for VSAM data sets 294
Choosing a data set type 295
Defining files for VSAM data sets 297
Specifying ENVIRONMENT options 298
Performance options 301
Defining files for alternate index paths 301
Defining VSAM data sets 302
Entry-sequenced data sets 302
Loading an ESDS 303
Using a SEQUENTIAL file to access an ESDS 303
Key-sequenced and indexed entry-sequenced data sets 306
Loading a KSDS or indexed ESDS 308
Using a SEQUENTIAL file to access a KSDS or indexed ESDS 310
Using a DIRECT file to access a KSDS or indexed ESDS 310
Updating a KSDS 312
Alternate indexes for KSDSs or indexed ESDSs 313
Relative-record data sets 320
Loading an RRDS 321
Using a SEQUENTIAL file to access an RRDS 324
Using a DIRECT file to access an RRDS . . . 324
Using files defined for non-VSAM data sets . . . 326
Using shared data sets 326

Part 3. Improving your program 327

Chapter 15. Improving performance 329
Selecting compiler options for optimal performance 329
OPTIMIZE 329
GONUMBER 329
ARCH 330
REDUCE 330
RULES 330
PREFIX 331
CONVERSION 332
FIXEDOVERFLOW 332
DEFAULT 332
Summary of compiler options that improve performance 335
Coding for better performance 335
DATA-directed input and output 336
Input-only parameters 336
GOTO statements 336
String assignments 337

Loop control variables 337
PACKAGEs versus nested PROCEDUREs . . . 338
REDUCIBLE functions 338
DESCLOCATOR or DESCLIST 339
DEFINED versus UNION 339
Named constants versus static variables . . . 340
Avoiding calls to library routines 341
Preloading library routines 342

Part 4. Using interfaces to other products 343

Chapter 16. Using the Sort program 345
Preparing to use Sort 345
Choosing the type of Sort 346
Specifying the sorting field 349
Specifying the records to be sorted 350
Determining storage needed for Sort 351
Calling the Sort program 352
Example 1 353
Example 2 354
Example 3 354
Example 4 354
Example 5 354
Determining whether the Sort was successful 355
Establishing data sets for Sort 355
Sort data input and output 356
Data input and output handling routines 356
E15—Input handling routine (Sort Exit E15) . . 357
E35—Output handling routine (Sort Exit E35) 360
Calling PLISRTA example 361
Calling PLISRTB example 361
Calling PLISRTC example 362
Calling PLISRTD example 363
Sorting variable-length records example . . . 365

Chapter 17. ILC with C 367
Equivalent data types 367
Simple type equivalence 367
Struct type equivalence 368
Enum type equivalence 368
File type equivalence 369
Using C functions 369
Matching simple parameter types 370
Matching string parameter types 373
Functions returning ENTRYs 374
Linkages 375
Sharing output and input 377
Sharing output 377
Sharing input 378
Using the ATTACH statement 378
Redirecting C standard streams 378
Summary 378

Chapter 18. Interfacing with Java . . . 381
Java Native Interface (JNI) 381
Calling PL/I program from Java 382
JNI sample program #1 - 'Hello World' 382
Step 1: Writing the Java program 382
Step 2: Compiling the Java program 383
Step 3: Writing the PL/I Program 383
Step 4: Compiling and linking the PL/I program 385
Step 5: Running the sample program 385
JNI sample program #2 - Passing a string 386
Step 1: Writing the Java program 386
Step 2: Compiling the Java program 387
Step 3: Writing the PL/I program 388
Step 4: Compiling and linking the PL/I program 390
Step 5: Running the sample program 390
JNI sample program #3 - Passing an integer . . . 390
Step 1: Writing the Java program 390
Step 2: Compiling the Java program 393
Step 3: Writing the PL/I program 393
Step 4: Compiling and linking the PL/I program 394
Step 5: Running the sample program 395
JNI sample program #4 - Java invocation API . . 395
Step 1: Writing the Java program 395
Step 2: Compiling the Java program 396
Step 3: Writing the PL/I program 396
Step 4: Compiling and linking the PL/I program 399
Step 5: Running the sample program 399
Attaching programs to an existing Java VM . . . 399
Determining equivalent Java and PL/I data types 400

Part 5. Specialized programming tasks 401

Chapter 19. Using the PLISAXA and PLISAXB XML parsers 403
Overview 403
The PLISAXA built-in subroutine 404
The PLISAXB built-in subroutine 404
The SAX event structure 405
start_of_document 405
version_information 406
encoding_declaration 406
standalone_declaration 406
document_type_declaration 406
end_of_document 406
start_of_element 406
attribute_name 406
attribute_characters 406
attribute_predefined_reference 407
attribute_character_reference 407
end_of_element 407
start_of_CDATA_section 407
end_of_CDATA_section 407
content_characters 407
content_predefined_reference 408
content_character_reference 408
processing_instruction 408
comment 408
unknown_attribute_reference 408
unknown_content_reference 408
start_of_prefix_mapping 408
end_of_prefix_mapping 408
exception 408
Parameters to the event functions 409
Coded character sets for XML documents 409

Supported EBCDIC code pages 410
Supported ASCII code pages 410
Specifying the code page 410
Exceptions 411
Example 412
Continuable exception codes 424
Terminating exception codes 428

Chapter 20. Using the PLISAXC and PLISAXD XML parsers 433
Overview. 433
The PLISAXC built-in subroutine. 434
The PLISAXD built-in subroutine. 434
The SAX event structure. 435
start_of_document. 436
version_information 436
encoding_declaration. 436
standalone_declaration 436
document_type_declaration. 436
end_of_document 436
start_of_element 436
attribute_name 436
attribute_characters 436
end_of_element. 437
start_of_CDATA_section. 437
end_of_CDATA_section 437
content_characters. 437
processing_instruction 437
comment 438
namespace_declare 438
end_of_input 438
unresolved_reference 438
exception. 438
Parameters to the event functions 438
Differences in the events 441
Coded character sets for XML documents 442
Supported code pages 442
Specifying the code page 442
Exceptions 443
Parsing XML documents with validation 443
XML schema 444
Creating an OSR 445
Example with a simple document 445
Example of using the PLISAXC built-in subroutine 445
Example of using the PLISAXD built-in subroutine 455

Chapter 21. Using PLIDUMP 467
PLIDUMP usage notes 468
Locating variables in the PLIDUMP output . . . 469
Locating AUTOMATIC variables 469
Locating STATIC variables 470
Locating CONTROLLED variables 471
Saved compilation data 475
Copyright 475
Timestamp 475
Saved options string 476

Chapter 22. Interrupts and attention processing 477
Using ATTENTION ON-units 478
Interaction with a debugging tool 478

Chapter 23. Using the Checkpoint/Restart facility. 479
Requesting a checkpoint record 479
Defining the checkpoint data set 480
Requesting a restart 481
Automatic restart after a system failure. . . . 481
Automatic restart within a program 481
Getting a deferred restart 481
Modifying checkpoint/restart activity 482

Chapter 24. Using user exits 483
Procedures performed by the compiler user exit 483
Structure of global control blocks. 484
The IBM-supplied compiler exit, IBMUEXIT . . . 485
Activating the compiler user exit 486
Customizing the compiler user exit 486
Modifying SYSUEXIT 486
Writing your own compiler exit 487
Writing the initialization procedure 487
Writing the message filtering procedure . . . 487
Writing the termination procedure 489
Example of suppressing SQL messages 490

Chapter 25. PL/I descriptors 497
Passing an argument 497
Argument passing by descriptor list. 497
Argument passing by locator/descriptor . . . 498
CMPAT(V*) descriptors 498
String descriptors 498
Array descriptors 500
CMPAT(LE) descriptors 500
String descriptors 501
Array descriptors 501

Part 6. Appendixes 503

Appendix. SYSADATA message information 505
Understanding the SYSADATA file 505
Summary record 506
Options record 507
Counter records 507
Literal records 507
File records 508
Message records 508
Understanding SYSADATA symbol information 509
Ordinal type records 509
Ordinal element records. 510
Symbol records. 511
Understanding SYSADATA syntax information . . 514
Source records 514
Token records 515
Syntax records 516

Notices 525
Trademarks 526

Bibliography 527
PL/I publications 527
Related publications 527

Glossary 529

Index 547

Tables

1. How to use Enterprise PL/I publications xv
2. How to use z/OS Language Environment publications xvi
3. Compile-time options, abbreviations, and IBM-supplied defaults 4
4. Supported CCSIDs 16
5. SYSTEM option table 87
6. Using the FLAG option to select the lowest message severity listed 110
7. Description of PL/I error codes and return codes 111
8. SQL preprocessor options and IBM-supplied defaults 126
9. SQL data types generated from PL/I declarations 136
10. SQL data types generated from SQL TYPE declarations 137
11. SQL data types mapped to PL/I declarations 137
12. SQL data types mapped to SQL TYPE declarations 138
13. Compile-time option flags supported by Enterprise PL/I under z/OS UNIX 161
14. Compiler standard data sets 162
15. Attributes of PL/I file declarations 228
16. A comparison of data set types available to PL/I record I/O 236
17. Information required when you create a library 240
18. Statements and options allowed for creating and accessing consecutive data sets 267
19. IBM machine code print control characters (CTL360) 271
20. Effect of LEAVE and REREAD Options 271
21. Creating a consecutive data set with record I/O: essential parameters of the DD statement 272
22. Accessing a consecutive data set with record I/O: essential parameters of the DD statement 274
23. Statements and options allowed for creating and accessing regional data sets 280
24. Creating a regional data set: essential parameters of the DD statement 288
25. DCB subparameters for a regional data set 289
26. Accessing a regional data set: essential parameters of the DD statement 289
27. Types of VSAM data sets and corresponding PL/I data set organization 292
28. Types and advantages of VSAM data sets 294
29. VSAM data sets and allowed file attributes 296
30. Processing allowed on alternate index paths 297
31. Statements and options allowed for loading and accessing VSAM entry-sequenced data sets 302
32. Statements and options allowed for loading and accessing VSAM indexed data sets . . . 306
33. Statements and options allowed for loading and accessing VSAM relative-record data sets. 320
34. The entry points and arguments to PLISRT*x* (*x* = A, B, C, or D) 352
35. C and PL/I type equivalents 367
36. Java primitive types and PL/I native equivalents 400
37. Continuable exceptions 424
38. Terminating exceptions 428

© Copyright IBM Corp. 1999, 2017

Figures

1. Including source statements from a library 102
2. Finding statement number (compiler listing example) 107
3. Finding statement number (runtime message example) 107
4. Compiler listing example 112
5. Using the macro preprocessor to produce a source deck 122
6. The PL/I declaration of SQLCA 130
7. The PL/I declaration of an SQL descriptor area 131
8. SQL statement containing indicator variables 140
9. pliclob sample program 143
10. Invoking a cataloged procedure 150
11. Cataloged Procedure IBMZC 151
12. Cataloged procedure IBMZCB 152
13. Cataloged procedure IBMZCBG 154
14. Declaration of PLITABS 169
15. PAGELENGTH and PAGESIZE 169
16. Output with automatic prompt 170
17. Output with no automatic prompt 170
18. Sample JCL to compile, link, and invoke the user exit 176
19. Sample program to display program arguments from the CPPL under TSO when using SYSTEM(STD) option 184
20. Sample program to display z/OS UNIX arguments and environment variables . . . 185
21. Output with automatic prompt 188
22. Output with no automatic prompt 189
23. Sample program to display program arguments from the CPPL under TSO when using SYSTEM(STD) option 192
24. Sample program to display z/OS UNIX arguments and environment variables . . . 193
25. Fixed-length records 222
26. How the operating system completes the DCB 227
27. Creating new libraries for compiled object modules 242
28. Placing a load module in an existing library 242
29. Creating a library member in a PL/I program 243
30. Updating a library member 243
31. Creating a data set with stream-oriented data transmission 251
32. Writing graphic data to a stream file 252
33. Accessing a data set with stream-oriented data transmission 254
34. Creating a print file via stream data transmission 257
35. PL/I structure PLITABS for modifying the preset tab settings 258
36. American National Standard print and card punch control characters (CTLASA) 270
37. Merge Sort—creating and accessing a consecutive data set 275
38. Printing record-oriented data transmission 277
39. Creating a REGIONAL(1) data set 284
40. Updating a REGIONAL(1) data set 286
41. Defining and loading an ESDS 305
42. Updating an ESDS 306
43. Defining and loading a key-sequenced data set (KSDS) 309
44. Updating a KSDS 311
45. Creating a unique key alternate index path for an ESDS 313
46. Creating a nonunique key alternate index path for an ESDS 314
47. Creating a unique key alternate index path for a KSDS 315
48. Alternate index paths and backward reading with an ESDS 317
49. Using a unique alternate index path to access a KSDS 319
50. Defining and loading a relative-record data set (RRDS) 323
51. Updating an RRDS. 325
52. Flow of control for Sort program 348
53. Flowcharts for input and output handling subroutines 358
54. Skeletal code for an input procedure 359
55. Skeletal code for an output handling procedure 360
56. PLISRTA—sorting from input data set to output data set 361
57. PLISRTB—sorting from input handling routine to output data set 362
58. PLISRTC—sorting from input data set to output handling routine 363
59. PLISRTD—sorting from input handling routine to output handling routine 364
60. Sorting varying-length records using input and output handling routines 365
61. Simple type equivalence 367
62. Sample struct type equivalence 368
63. Sample enum type equivalence 369
64. Start of the C declaration for its FILE type 369
65. PL/I equivalent for a C file 369
66. Sample code to use fopen and fread to dump a file 370
67. Declarations for filedump program 370
68. C declaration of fread 371
69. First incorrect declaration of fread 371
70. Second incorrect declaration of fread 371
71. Third incorrect declaration of fread 371
72. Code generated for RETURNS BYADDR 372
73. Correct declaration of fread 372
74. Code generated for RETURNS BYVALUE 372
75. First incorrect declaration of fopen 373
76. Second incorrect declaration of fopen 373
77. Correct declaration of fopen 373
78. Optimal, correct declaration of fopen 373

© Copyright IBM Corp. 1999, 2017

79. Declaration of fclose 374
80. Commands to compile and run filedump 374
81. Output of running filedump 374
82. Sample compare routine for C qsort function 374
83. Sample code to use C qsort function 375
84. Incorrect declaration of qsort 375
85. Correct declaration of qsort 375
86. Code when parameters are BYADDR 376
87. Code when parameters are BYVALUE 377
88. Java sample program #2 - Passing a string 387
89. PL/I sample program #2 - Passing a string 389
90. Java sample program #3 - Passing an integer 392
91. PL/I sample program #3 - Passing an integer 394
92. Java sample program #4 - Receiving and printing a string 395
93. PL/I sample program #4 - Calling the Java invocation API 398
94. Sample XML document 405
95. PLISAXA coding example - type declarations 413
96. PLISAXA coding example - event structure 414
97. PLISAXA coding example - main routine 415
98. PLISAXA coding example - event routines 416
99. PLISAXA coding example - program output 424
100. Sample XML document 435
101. PLISAXC coding example - type declarations 446
102. PLISAXC coding example - event structure 447
103. PLISAXC coding example - main routine 448
104. PLISAXC coding example - event routines 449
105. PLISAXC coding example - program output 455
106. PLISAXD coding example - event routines 456
107. Output from PLISAXD sample 466
108. Example PL/I routine calling PLIDUMP 467
109. Using an ATTENTION ON-unit 478
110. PL/I compiler user exit procedures 484
111. Example of an user exit input file 486
112. Suppressing SQL messages 490
113. Record types encoded as an ordinal value 506
114. Declare for the header part of a record 506
115. Declare for a summary record 507
116. Declare for a counter record 507
117. Declare for a literal record 508
118. Declare for a file record 508
119. Declare for a message record 509
120. Declare for an ordinal type record 510
121. Declare for an ordinal element record 511
122. Symbol indices assigned to the elements of a structure 512
123. Data type of a variable 513
124. Declare for a source record 515
125. Declare for a token record 515
126. Declare for the token record kind 516
127. Node indices assigned to the blocks in a program 516
128. Declare for a syntax record 517
129. Declare for the syntax record kind 520
130. Node indices assigned to the syntax records in a program. 521
131. Declare for the expression kind 522
132. Declare for the number kind 522
133. Declare for the lexeme kind 523

Introduction

About this document

This book is for PL/I programmers and system programmers. It helps you understand how to use Enterprise PL/I for z/OS® Enterprise PL/I for AIX in order to compile PL/I programs. It also describes the operating system features that you might need to optimize program performance or handle errors.

Important: Enterprise PL/I for z/OS is referred to as Enterprise PL/I throughout this book.

Runtime environment for Enterprise PL/I for z/OS

Enterprise PL/I uses Language Environment as its runtime environment. It conforms to the Language Environment architecture and can share the runtime environment with other Language Environment-conforming languages.

Language Environment provides a common set of runtime options and callable services. It also improves interlanguage communication (ILC) between high-level languages (HLL) and the assembler by eliminating language-specific initialization and termination on each ILC invocation.

Using your documentation

The publications provided with Enterprise PL/I are designed to help you program with PL/I. The publications provided with Language Environment are designed to help you manage your runtime environment for applications generated with Enterprise PL/I. Each publication helps you perform a different task.

The following tables show you how to use the publications you receive with Enterprise PL/I and Language Environment. You will want to know information about both your compiler and runtime environment. For the complete titles and order numbers of these and other related publications, see "Bibliography" on page 527.

PL/I information

Table 1. How to use Enterprise PL/I publications

To...	Use...
Evaluate Enterprise PL/I	Fact Sheet
Understand warranty information	Licensed Program Specifications
Plan for and install Enterprise PL/I	Enterprise PL/I Program Directory
Understand compiler and runtime changes and adapt programs to Enterprise PL/I and Language Environment	Compiler and Run-Time Migration Guide
Prepare and test your programs and get details on compiler options	Programming Guide
Get details on PL/I syntax and specifications of language elements	Language Reference

© Copyright IBM Corp. 1999, 2017

Table 1. How to use Enterprise PL/I publications (continued)

To...	Use...
Diagnose compiler problems and report them to IBM®	Diagnosis Guide
Get details on compile-time messages	Compile-Time Messages and Codes

Language Environment information

Table 2. How to use z/OS Language Environment publications

To...	Use...
Evaluate Language Environment	Concepts Guide
Plan for Language Environment	Concepts Guide Runtime Application Migration Guide
Install Language Environment on z/OS	z/OS Program Directory
Customize Language Environment on z/OS	Customization
Understand Language Environment program models and concepts	Concepts Guide Programming Guide
Find syntax for Language Environment runtime options and callable services	Programming Reference
Develop applications that run with Language Environment	Programming Guide and your language Programming Guide
Debug applications that run with Language Environment, get details on runtime messages, diagnose problems with Language Environment	Debugging Guide and Run-Time Messages
Develop interlanguage communication (ILC) applications	Writing Interlanguage Applications
Migrate applications to Language Environment	Runtime Application Migration Guide and the migration guide for each Language Environment-enabled language

Notation conventions used in this document

This book uses the conventions, diagramming techniques, and notation described in "Conventions used" and "How to read the notational symbols" on page xix to illustrate PL/I and non-PL/I programming syntax.

Conventions used

Some of the programming syntax in this document uses type fonts to denote different elements:

- Items shown in UPPERCASE letters indicate key elements that must be typed exactly as shown.
- Items shown in lowercase letters indicate user-supplied variables for which you must substitute appropriate names or values. The variables begin with a letter and can include hyphens, numbers, or the underscore character (_).
- The term *digit* indicates that a digit (0 through 9) should be substituted.
- The term *do-group* indicates that a do-group should be substituted.
- Underlined items indicate default options.
- `Examples` are shown in monocase type.

- Unless otherwise indicated, separate repeatable items from each other by one or more blanks.

Note: Any symbols shown that are not purely notational, as described in "How to read the notational symbols" on page xix, are part of the programming syntax itself.

For an example of programming syntax that follows these conventions, see "Example of notation" on page xix.

How to read the syntax notation

The following rules apply to the syntax diagrams used in this document:

Arrow symbols

Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

►►— Indicates the beginning of a statement.

—► Indicates that the statement syntax is continued on the next line.

►— Indicates that a statement is continued from the previous line.

—►◄ Indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the ►— symbol and end with the —► symbol.

Conventions

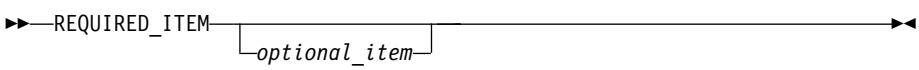
- Keywords, their allowable synonyms, and reserved parameters, appear in uppercase for MVS™ and OS/2 platforms, and lowercase for UNIX platforms. These items must be entered exactly as shown.
- Variables appear in lowercase italics (for example, *column-name*). They represent user-defined parameters or suboptions.
- When entering commands, separate parameters and keywords by at least one blank if there is no intervening punctuation.
- Enter punctuation marks (slashes, commas, periods, parentheses, quotation marks, equal signs) and numbers exactly as given.
- Footnotes are shown by a number in parentheses, for example, (1).
- A ⌂ symbol indicates one blank position.

Required items

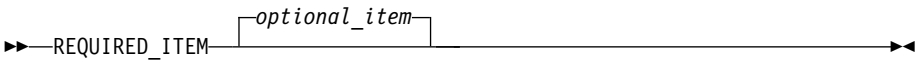
Required items appear on the horizontal line (the main path).

Optional Items

Optional items appear below the main path.

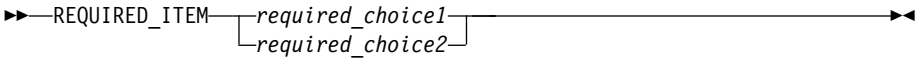


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

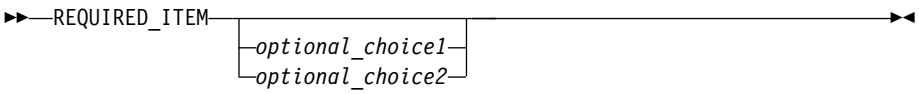


Multiple required or optional items

If you can choose from two or more items, they appear vertically in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

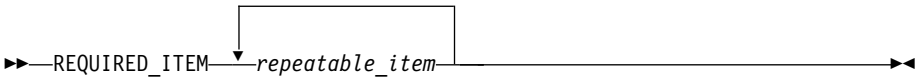


If choosing one of the items is optional, the entire stack appears below the main path.

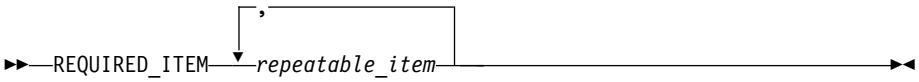


Repeatable items

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



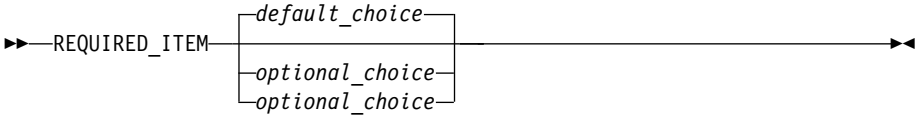
If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can specify more than one of the choices in the stack.

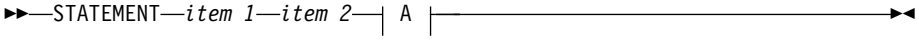
Default keywords

IBM-supplied default keywords appear above the main path, and the remaining choices are shown below the main path. In the parameter list following the syntax diagram, the default choices are underlined.



Fragments

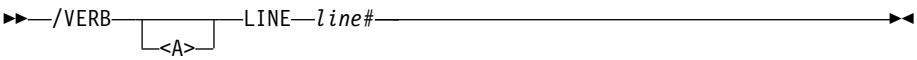
Sometimes a diagram must be split into fragments. The fragments are represented by a letter or fragment name, set off like this: `| A |`. The fragment follows the end of the main diagram. The following example shows the use of a fragment.



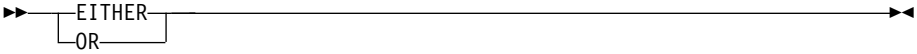
A:

Substitution-block

Sometimes a set of several parameters is represented by a substitution-block such as **<A>**. For example, in the imaginary /VERB command you could enter `/VERB LINE 1`, `/VERB EITHER LINE 1`, or `/VERB OR LINE 1`.



where <A> is:



Parameter endings

Parameters with number values end with the symbol '#', parameters that are names end with 'name', and parameters that can be generic end with '*'.

The MSNAME keyword in the example supports a name value and the SYSID keyword supports a number value.

How to read the notational symbols

Some of the programming syntax in this book is presented using notational symbols. This is to maintain consistency with descriptions of the same syntax in other IBM publications, or to allow the syntax to be shown on single lines within a table or heading.

- **Braces**, { }, indicate a choice of entry. Unless an item is underlined, indicating a default, or the items are enclosed in brackets, you must choose at least one of the entries.
- Items separated by a single **vertical bar**, |, are alternative items. You can select only one of the group of items separated by single vertical bars. (Double vertical bars, ||, specify a concatenation operation, not alternative items. See the *PL/I Language Reference* for more information on double vertical bars.)
- Anything enclosed in **brackets**, [ ], is optional. If the items are vertically stacked within the brackets, you can specify only one item.
- An **ellipsis**, ..., indicates that multiple entries of the type immediately preceding the ellipsis are allowed.

Example of notation

The following example of PL/I syntax illustrates the notational symbols described in "How to read the notational symbols":

```
DCL file-reference FILE STREAM
                  {INPUT | OUTPUT [PRINT]}
                  ENVIRONMENT(option ...);
```

Interpret this example as follows:

- You must spell and enter the first line as shown, except for *file-reference*, for which you must substitute the name of the file you are referencing.
- In the second line, you can specify INPUT or OUTPUT, but not both. If you specify OUTPUT, you can optionally specify PRINT as well. If you do not specify either alternative, INPUT takes effect by default.
- You must enter and spell the last line as shown (including the parentheses and semicolon), except for *option ...*, for which you must substitute one or more options separated from each other by one or more blanks.

Summary of changes

Enhancements in this release

This release provides the following functional enhancements that are described in this and the other IBM PL/I books.

Changes in GI13-4536-00, April 2017

Enhancements in usability:

- The compiler now flags unreachable ASSERT UNREACHABLE statements with a different message than it flags other unreachable statements.
- The compiler now expands in the AGGREGATE listing typed structures that are member of other structures.
- The attributes listing now shows the contents of the VALUE attribute for CHARACTER and BIT constants of length 256 or less and also for numeric PICTURE constants

Compiler option enhancements:

- The new "ASSERT" on page 9 compiler option controls whether ASSERT statements call a default library routine that will raise the ASSERTION condition or a routine provided by the user.
- The new "CASE" on page 12 compiler option controls whether some names will be shown in uppercase or in the same format as they appear in the source program.
- The CMPAT(LE) and CMPAT(V1) have been restored in the "CMPAT" on page 15 compiler option.
- The "RULES" on page 70 compiler option has these new suboptions:

  LAXINTERFACE | NOLAXINTERFACE
  You can use RULES(NOLAXINTERFACE) to flag code that does not contain a valid explicit declare for each of its external PROCEDUREs.

  MULTIENTRY | NOMULTIENTRY
  You can use RULES(NOMULTIENTRY) to flag code that contains ENTRY statements.

  MULTIEXIT | NOMULTIEXIT
  You can use RULES(NOMULTIEXIT) to flag code that contains multiple RETURN statements.

UNREFCTL | NOUNREFCTL
You can use RULES(NOUNREFCTL) to flag unreferenced CTL variables.

UNREFDEFINED | NOUNREFDEFINED
You can use RULES(NOUNREFDEFINED) to flag unreferenced DEFINED variables.

UNREFENTRY | NOUNREFENTRY
You can use RULES(NOUNREFENTRY) to flag unreferenced ENTRY constants.

UNREFFILE | NOUNREFFILE
You can use RULES(NOUNREFFILE) to flag unreferenced FILE constants.

UNREFSTATIC | NOUNREFSTATIC
You can use RULES(NOUNREFSTATIC) to flag unreferenced STATIC variables.

YY | NOYY
You can use RULES(NOYY) to flag the use of 2-digit years.

- The NOELSEIF suboption of the "RULES" on page 70 compiler option now also flag ELSE statements immediately followed by an IF statement that is enclosed in a simple DO-END.

CICS enhancements:

- The CICS preprocessor output now includes a listing of all the CICS options in effect when the preprocessor run.

Performance improvements

- Enterprise PL/I now takes additional exploitation of the vector facility.
- Some fixed decimal divides with large precision are now done using the Decimal Floating-Point (DFP) facility. This might cause some ZERODIVIDE exceptions to be reported as INVALIDOP.

Usability enhancements

- Enterprise PL/I now provides support for compiling code for 64-bit applications. For more information, see Chapter 7, "Considerations for developing 64-bit applications," on page 195.
- The compiler now issues a warning message if the option strings in the IBMZIOP are the same as each other and if a user specifies an option that conflicts with one of those non-overridable options.
- The compiler now also includes information about the compiler options that are used during compilation in the generated SYSADATA file. All the declarations for SYSADATA records are now provided in include files in the samples data set SIBMZSAM.

Compiler option enhancements

- The new "BRACKETS" on page 12 compiler option allows you to specify the symbols that the SQL preprocessor accepts as the left and right brackets in SQL array references. This option makes it easier to use such language.
- The new "DECOMP" on page 22 compiler option causes the compiler to produce a listing section that shows all intermediate expressions and their attributes for all expressions used in the source program.

- The "DECIMAL" on page 20 compiler option has a new suboption, TRUNCFLOAT, which provides you with finer control over how the compiler handles assignments of float to fixed decimal when truncation might occur.
- The new "EXPORTALL" on page 34 compiler option controls whether to export all externally defined procedures and variable so that a DLL application can use them.
- The new "HEADER" on page 39 compiler option lets you control what appears in the middle of each header line in the compiler listing.
- The "INSOURCE" on page 43 compiler option has two new suboptions, FIRST and ALL, which control how many source listings appear in the listing file.
- The new "LP" on page 49 compiler option allows you to specify whether the compiler generates 31-bit code or 64-bit code.
- The "MDECK" on page 54 compiler option has two new suboptions, AFTERALL and AFTERMACRO, which control when the MDECK is generated.
- The new "NULLDATE" on page 57 compiler option allows you to use the SQL null date as a valid date in your program.
- The new "OFFSETSIZE" on page 58 compiler option determines the size of OFFSET variables in 64-bit applications.
- The "RULES" on page 70 compiler option has these new suboptions:

  LAXSTMT | NOLAXSTMT
  : You can use RULES(NOLAXSTMT) to flag any line that has more than one statement.

  UNREFBASED | NOUNREFBASED
  : You can use RULES(NOLAXINTERFACE) to flag unreferenced BASED variables that are in BASED storage.

- The NOPROCENDONLY suboption of the "RULES" on page 70 compiler option has two new suboptions, ALL and SOURCE. They make it easier to stage enforcement of the rules that the END statement for every procedure must include the name of that procedure.
- The "XREF" on page 97 compiler option has two new suboptions, EXPLICIT and IMPLICIT, which allow you to determine whether implicit variable references are included in the XREF listing.

Note:

All the compiler option default settings for a release is listed in the member with the name IBMXO*vrm*, where *v* is the version number, *r* is the release number and *m* is the modification number which is usually 0 (for example, IBMXO510 for V5R1 and IBMXO450 for V4R5), in the SIBMZSAM data set. The data set contains IBMXO*vrm* members for all supported PL/I releases. To see what has changed in the default settings and what new options are added from one release to the next, you can compare the IBMXO*vrm* files for the two releases. You can also use the IBMXO*vrm* files as templates for creating your own options files with your preferred settings.

SQL enhancements

- The SQL preprocessor now parses the DEFINE ALIAS, DEFINE ORDINAL, and DEFINE STRUCTURE statements. If a DEFINE ALIAS statement defines a PL/I type that can be used in SQL statements, a variable declared with that type can also be used in SQL statements.

- If the "NULLDATE" on page 57 compiler option is specified, the SQL null date, with *year*, *month*, and *day* all equal to 1, is accepted as a valid date in some of the datetime handling built-in functions.
- The INDFOR attribute makes it easy to declare a structure of indicator variables to match another PL/I structure.
- To make it easier to change the severity of the DSNH030 SQL preprocessor message, it is now given its own preprocessor message IBM3317.

Enhancements from V4R5

This release provides the following functional enhancements that are described in this and the other IBM PL/I books.

Performance improvements

- The code that is generated for the EXEC CICS® statement executes faster because one MVC is eliminated.
- Much faster code is generated for MOD and REM of FIXED DEC with precision greater than 15.
- The ARCH option now accepts 11 as its maximum value, and when ARCH(11) is specified, the compiler generates code that exploits the new hardware instructions on those z systems. This code especially improves the performance of some instances of the SEARCH, and VERIFY built-in functions.

SQL enhancements

- The validation of an EXEC SQL statement will not stop when the first invalid host variable is found, but will instead check all host variable references.
- The new SQL preprocessor option (NO)CODEPAGE determines how the compiler CODEPAGE option is honored by the SQL preprocessor.
- The new SQL preprocessor option (NO)WARNDECP allows you to reduce the amount of "noise" that is produced by the SQL preprocessor.
- A structure that is used as a host variable can now also have a structure as its indicator variable.
- A named constant, that is, a PL/I variable with the VALUE attribute can now be used as a host variable if SQL allows a constant in that setting.

Usability enhancements

- The new MAXBRANCH compiler option can help you find excessively complex code.
- The new FILEREF and NOFILEREF compiler options control whether the compiler produces a file reference table.
- The new JSON compiler option lets you choose the case of the names in the JSON text generated by the JSONPUT built-in functions and expected by the JSONGET built-in functions.
- The LIMITS compiler option now accepts STRING as a suboption with these values as the threshold for the length of a BIT, CHARACTER or WIDECHAR variable: 32K, 512K, 8M, and 128M.
- The XML compiler option now lets you choose whether XML attributes generated by the XMLCHAR built-in function are enclosed in apostrophes or quotation marks.
- The RULES(NOLAXRETURN) compiler option is enhanced so that ERROR will be raised if code falls through to the END statement in a PROCEDURE with the RETURNS attribute.

- The new ALL | SOURCE suboptions to the RULES(NOLAXNESTED) and RULES(NOPADDING) compiler options provide finer control over when the compiler flags questionable coding.
- The new FORCE(NOLAXQUAL) attribute and the new FORCE suboption of the RULES(NOLAXQUAL) option enable users to enforce the NOLAXQUAL rules in a structure-by-structure manner.
- The use of INITIAL on REFER objects will now be flagged with an E-level message.
- The MACRO preprocessor now supports DEPRECATE and DEPRECATENEXT options that allow you to remove selected macro procedures over a course of planned stages.
- If a FIXED or FLOAT variable has the VALUE attribute, the compiler lists its value as a character string in the ATTRIBUTES listing.
- The aggregate listing is enhanced to mark arrays of VARYING that contain padding between array members (for example, when a structure contains an array of CHAR(31) VARYING ALIGNED).
- When SUBSCRIPTRANGE is enabled and an array assignment includes arrays with non-constant bounds, the compiler now generates code that raises the SUBSCRIPTRANGE condition if the bounds do not match. If the bounds are all constant, the compiler continues to check at compile-time that they match.
- If an event in any of the PLISAX event structures is set to null (via the NULLENTRY built-in function or the UNSPEC pseudovariable), the XML parser will not call that event. This allows you to limit your XML parsing code to only the events in which you are interested and to improve the performance of the overall parse at the same time.
- The compiler now flags an attempt to FETCH or RELEASE the procedure that it is currently compiling.

Enhancements from V4R4

This release provides the following functional enhancements that are described in this and the other IBM PL/I books.

Performance improvements

- The generation of the pseudo assembler listing is faster than the previous releases.
- The compiler is now compiled with ARCH(7).
- The compiler now flags declarations for variables that do not have the STATIC attribute but have more than 100 INITIAL items.

Usability enhancements

- The SQL preprocessor now issues more messages when programs contain incorrect syntax. These messages make it easier to identify the source statement that is in error.
- The new NOINCLUDE option can be used to prohibit the use of the %INCLUDE and %XINCLUDE statements outside the MACRO preprocessor.
- The included files are now bracketed in the source listing when the %INCLUDE statement is processed by the final compiler pass.
- The NULLSTRPTR suboption of the DEFAULT option has the new STRICT suboption that flags the assignments and comparisons of '' to POINTERS as invalid.

- When the STMT option is specified, the source and message listings will include both the logical statement numbers and the source file numbers.

Enhancements from V4R3

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Performance improvements

- The ARCH(10) option provides exploitation of the IBM zEnterprise® EC12 System instructions.
- The compiler now generates inline code for VERIFY and SEARCH when they have three arguments and the second argument is a single character.
- The compiler now generates inline code for more conversions from PICTURE to DFP.
- The compiler now generates inline code for more conversions from BIT to CHAR.
- The compiler now generates inline code for conversions of BIT to WIDECHAR.
- The code generated for TRIM of FIXED DEC is improved.

Usability enhancements

- The SQL preprocessor has the following improvements:
  - It supports the ONEPASS option.
  - It supports the use of some restricted expressions in the host variable declarations.
  - The listing of the EXEC SQL statement is displayed in a readable format that is similar to the original source.
  - It supports the use of host variables declared with the LIKE attribute.
  - It supports the new DEPRECATE option that causes the preprocessor to flag a list of statements that you want to deprecate.
- The ADATA file now records the use of the following attributes, built-in functions, and statement:
  - The INONLY, INOUT, and OUTONLY attributes
  - The XMLATTR and XMLOMIT attributes
  - The ALLCOMPARE, UTF8, UTF8TOCHAR, and UTF8TOWCHAR built-in functions
  - The ASSERT statement
- With the new CASERULES option, you can specify case rules for PL/I keywords. For example, you can specify the rule that all keywords must be in uppercase.
- The DEPRECATE option has a new STMT suboption that causes the compiler to flag a list of statements that you want to deprecate.
- The new DEPRECATENEXT option allows for staged deprecation of functions.
- The IGNORE option has a new ASSERT suboption that instructs the compiler to ignore all ASSERT statements.
- The new (NO)MSGSUMMARY option controls whether the compiler adds a summary of all messages that are issued during the compilation into the listing.
- The RTCHECK option has the new NULL370 suboption that checks whether a pointer that equals to the old NULL() value is dereferenced. Pointers that equal to the old NULL() value are these with the hexadecimal value of 'FF000000'x.

- The RULES option now accepts (NO)CONTROLLED as a suboption that controls whether to flag the use of the CONTROLLED attribute.
- The RULES option now accepts (NO)LAXNESTED as a suboption that controls whether to flag programs where nested procedures exist between sections of executable code.
- The RULES option now accepts (NO)RECURSIVE as a suboption that determines whether to flag any use of the RECURSIVE attribute or any procedure that directly calls itself.
- The RULES(NOUNREF) option now accepts SOURCE | ALL as a suboption that determines whether to flag all unreferenced variables.
- RULES(NOLAXIF) now also causes the compiler to flag the assignments of the form x = y = z.
- RULES(NOLAXSCALE) now also causes the compiler to flag ROUND(x, p) when p < 0.

Enhancements from V4R2

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Performance improvements

- The ARCH(9) option provides further exploitation of the IBM zEnterprise 196 (z196) System instructions, including the high-word, floating-point extension, and population count facilities.
- The new UNROLL compiler option gives you control of loop unrolling.
- The compiler now generates inline code to resolve the ULENGTH and USUBSTR built-in functions for character strings.
- The compiler now generates inline code for MEMINDEX(*p*, *n*, *x*) where *x* is WCHAR(1).
- The compiler now generates inline code for STG(*x*) where *x* is BASED variable using REFER and meets both of the following conditions:
  - All NONVARYING BIT in x are specified with the ALIGNED attribute.
  - All other elements in x are specified with UNALIGNED.

Debugging improvements

- The compiler now supports typed structures in Debug Tool.

SQL support enhancements

- The SQL Preprocessor has the following significant changes:
  - It fully and correctly supports block scoping.
  - The preprocessor load module is more than eight times smaller.
  - The preprocessor runs faster.
  - It supports the use of the SQL TYPE attribute anywhere you can specify a PL/I data type.
  - It now supports the following compiler options when processing declarations of host variables. It applies the defaults correctly and rejects unsuitable host variables as appropriate.
    - DEFAULT(ANS | IBM)
    - DEFAULT(ASCII | EBCDIC)
    - DEFAULT((NO)EVENDEC)
    - DEFAULT((NON)NATIVE)

- DEFAUT(SHORT(HEX | IEEE))
- RULES((NO)LAXCTL)

– It now correctly processes the PRECISION attribute.
– It now recognizes the UNSIGNED and COMPLEX attributes and rejects the use of these attributes in any host variable.
– It now ensures that DSNHMLTR is declared in the outermost procedure that contains codes that need DSNHMLTR.
– It now handles packages correctly.
– It now ensures that characters in the source code are printable when it generates code to set the SQLAVDAID.
– It no longer requires that indicator arrays have a lower bound of 1.
– It now generates an SQL parameter list structure that has fewer unions, fewer init clauses, and no additional declarations based on elements of the structure.

Usability enhancements

- The new PPLIST compiler option conditionally erases the part of the listing that is generated by any preprocessor phase that produces no messages.
- The compiler issues better messages when a comma is missing in a structure declaration.
- The compiler issues new messages when the source code contains invalid shift-in and shift-out bytes.
- The compiler applies the NONASSIGNABLE attribute to any parameter that is declared with the INONLY attribute; it flags any assignment to a parameter that is declared as INONLY.
- Under RULES(NOLAXENTRY), the compiler does not flag names that start with DSN.
- Under RULES(NOUNREF), the compiler does not flag names that start with DSN or SQL.
- Under the new NOSELFASSIGN suboption of the RULES compiler option, the compiler flags assignments of variables to themselves.
- Under the new NOLAXRETURN suboption of the RULES compiler option, the compiler generates codes to raise the ERROR condition when a RETURN statement is used in some invalid ways.

Enhancements from V4R1

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Debugging improvements

- Under TEST(SEPARATE), the compiler optionally places the statement number table in the debug file and thus reduces the size of the generated object deck.
- Under TEST(SEPARATE), the compiler includes information identifying the source lines for declarations, references, and assignments.
- Under TEST(SEPARATE), the compiler generates information to identify the implicit locator reference when a BASED variable is based on the ADDR of an array element or other complex references.

Performance improvements

- The ARCH(9) option provides exploitation of the IBM zEnterprise System instructions.

- If all the elements of structures using REFER are byte-aligned, the compiler inlines code to resolve references to these elements rather than through library calls.

Usability enhancements

- The new DEPRECATE compiler option flags variable names and included file names that you want to deprecate.
- The new SEPARATE suboption of the GONUMBER option is provided to place the generated statement number table in the separate debug file.
- The new NOGLOBALDO suboption of the RULES option flags all DO loop control variables that are declared in the parent block.
- The new NOPADDING suboption of the RULES option flags all structures that contain padding.
- The SQL preprocessor now supports the XREF option.
- The new PLISAXD built-in subroutine provides the ability to parse XML documents with validation against a schema by using the XML System Services parser.

Serviceability enhancements

- The use of either the GOSTMT option or the IMPRECISE option is now flagged as unsupported.
- The compiler now always lists all SQL preprocessor options that are in effect.

Enhancements from V3R9

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Performance improvements

- UVALID will now be inlined for for strings of length 256 or less.
- Under ARCH(7) and higher, the CU12, CU14, CU21, CU24, CU41, and CU42 instructions will be used to provide for fast conversions between UTF-8, UFT-16 and UTF-32.
- Under ARCH(7) and higher, the TRTT, TROT, TRTO, and TROO instructions will be used to provide for fast translations between one- and two-byte buffers.
- Assignments of like arrays of scalars will now be handled as storage copy operations.
- All assignments of BIT VARYING to BIT VARYING will now be inlined.
- All assignments of byte-aligned BIT NONVARYING to BIT VARYING will now be inlined.
- The ROUND and ROUNDDEC built-in functions will be inlined when the argument to be rounded is DFP.

  Note: From PL/I for z/OS V5.1, the ROUNDDEC built-in function has been renamed as ROUNDAWAYFROMZERO.
- To simplify the choosing of options for best performance,
  - the COMPACT option has been dropped
  - The default setting for DEFAULT(REORDER | ORDER) has been changed to DEFAULT(REORDER)
  - the TUNE option has been dropped

- Detection of the dereferencing of null pointers exploits the new compare-and-trap instruction under ARCH(8).
- The compiler has been built with ARCH(6) to improve its performance.

Usability enhancements

- The CICS preprocessor now supports block-scoping and consequently will add the needed local CICS declares to all non-nested procedures.
- The SQL preprocessor now supports the PL/I rules for the scope of declarations when resolving host variable references through the new SCOPE option. NOSCOPE is the default for compatibility with previous releases.
- The MACRO preprocessor will now leave %include, %xinclude, %inscan, and %xinscan statements in the compiler listing as comments.
- The MACRO preprocessor now provides via the %DO SKIP; statement an easy and clear way to omit sections of code from a compile.
- The MACRO preprocessor now supports an option called NAMEPREFIX which allows the user to force macro procedures and variables to start with a specified character.
- The IGNORE compiler option provides the ability to suppress PUT FILE and/or DISPLAY statements (either of which may have been used for debug purposes but which should be compiled out of the production version).
- The NULLSTRPTR suboption of the DEFAULT compiler option provides user control of whether sysnull or null is assigned to a pointer when the source in the assignment is a null string.
- The new MAXGEN option specifies the maximum number of intermediate language statements that should be generated for any one user statement and will cause the compiler to flag any statement where this maximum is exceeded.
- The new ONSNAP option will allow the user to request the compiler to insert an ON STRINGRANGE SNAP; or an ON STRINGSIZE SNAP; statement into the prologue of a MAIN or FROMALIEN proc.
- The new SHORT suboption of the INITAUTO option will limit the INITAUTO option so that it does not duplicate all of the runtime STORAGE option, but does initialize variables that might be optimized to registers.
- The new RTCHECK option will generate code to test for the dereferencing of null pointers.
- The compiler will now flag various statements that may be risky:
  - code where the result of a FIXED operation has a scale factor less than zero
  - ENTRYs used as functions but declared without the RETURNS attribute
  - parameters declared as BYVALUE when doing so is ill-advised, e.g. declaring a FIXED DEC parameter BYVALUE
  - FIXED DECIMAL add and multiply operations that might raise FIXEDOVERFLOW
- The RULES option has been expanded to allow more control over and flagging of poor code:
  - NOPROCENDONLY will flag END statements for PROCs that don't name the PROC they are closing
  - NOSTOP will flag the use of STOP and EXIT
  - NOLAXQUAL(STRICT) will flag variables not qualified with their level-1 name
  - NOLAXSCALE will flag declares of FIXED DEC(p,q) and FIXED BIN(p,q) where $q < 0$ or $p < q$

- NOGOTO(LOOSE) will allow GOTOs only if in the same block
- More than one DELAY STATEMENTS can be concurrently executed in different procedures.

Serviceability enhancements

- When the compiler cannot open a file, the compiler will now, if possible, also include the related C runtime message in the message in the listing.
- If user code requires a DFP conversion at compile time but the compile is running on a machine without DFP hardware, this error will be trapped and a meaningful error issued.
- If the SQL preprocessor is invoked more than once without INCONLY as its suboption, then the DBRM library created by the compiler will be empty, and now an E-level message will be issued to warn the user about this problem.

Enhancements from V3R8

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Performance improvements

- The ARCH(8) and TUNE(8) options provide exploitation of the z/HE instructions.
- The HGPR option supports using 64-bit registers in 32-bit code.
- The GOFF option supports generation of GOFF objects.
- The PFPO instruction will be exploited in conversions between differing float formats.
- SRSTU will be used in code generated for UTF-16 INDEX.
- Calls to null internal procedures will now be completely removed.

Usability enhancements

- PLISAXC provides for access to the XML System Services parser via a SAX interface.
- The INCDIR option is supported under batch.
- The LISTVIEW option now provides the support previously provided by the AFTERMACRO etc suboptions of TEST.
- The NOLAXENTRY suboption of the RULES option allows for the flagging of unprototyped ENTRYs.
- The (NO)FOFLONMULT suboption of the DECIMAL option allows for control of whether FOFL is raised in MULTIPLY of FIXED DECIMAL.
- The HEX and SUBSTR suboptions of the USAGE option provide more user control of the behavior of the corresponding built-in functions.
- The DDSQL compiler option provides the ability to specify an alternate DD name to be used for EXEC SQL INCLUDEs.
- The INCONLY suboption provided by the MACRO and SQL preprocessors can request those preprocessors to perform only INCLUDEs.
- The integrated SQL preprocessor will now generate DB2® precompiler style declares for all *LOB_FILE, *LOCATOR, ROWID, BINARY and VARBINARY SQL types, in addition to the BLOB, CLOB and DBCLOB SQL types already supported, when the LOB(DB2) SQL preprocessor option is selected.

Enhancements from V3R7

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Debugging improvements

- The TEST option has been enhanced so that users can choose to view the source in the listing and in the Debug Tool source window as that source would appear after a user-specified preprocessor had been run (or after all the preprocessors had been run).

Performance improvements

- The BASR instruction will now be used instead of the BALR instruction.
- The conversions of FIXED DEC with large precision to FLOAT will be inlined and speeded up by the use of FIXED BIN(63) as an intermediary.
- The CHAR built-in when applied to CHAR expressions will now always be inlined.
- The code generated for conversions of FIXED BIN(p,q) to unscaled FIXED DEC has been significantly improved.
- TRTR will be used, under ARCH(7), for SEARCHR and VERIFYR in the same situations where TRT would be used for SEARCH and VERIFY.
- UNPKU will be used to convert some PICTURE to WIDECHAR (rather than making a library call).

Usability enhancements

- IEEE Decimal Floating-Point (DFP) is supported.
- The new MEMCONVERT built-in function will allow the user to convert arbitrary lengths of data between arbitrary code pages.
- The new ONOFFSET built-in function will allow the user to have easy access to another piece of information formerly available only in the runtime error message or dump, namely the offset in the user procedure at which a condition was raised.
- The new STACKADDR built-in function will return the address of the current dynamic save area (register 13 on z/OS) and will make it easier for users to write their own diagnostic code.
- The length of the mnemonic field in the assembler listing will be increased to allow for better support of the new z/OS instructions that have long mnemonics.
- More of the right margin will be used in the attributes, cross-reference and message listings.
- The CODEPAGE option will now accept 1026 (the Turkish code page) and 1155 (the 1026 code page plus the Euro symbol).
- The new MAXNEST option allows the user to flag excessive nesting of BEGIN, DO, IF and PROC statements.
- Under the new (and non-default) suboption NOELSEIF of the RULES option, the compiler will flag any ELSE statement that is immediately followed by an IF statement and suggest that it be rewritten as a SELECT statement.
- Under the new (and non-default) suboption NOLAXSTG of the RULES option, the compiler will flag declares where a variable A is declared as BASED on ADDR(B) and STG(A) > STG(B) not only (as the compiler did before) when B is AUTOMATIC, BASED or STATIC with constant extents but now also when B is a parameter declared with constant extents.

- The new QUOTE option will allow the user to specify alternate code points for the quote (") symbol since this symbol is not code-page invariant.
- The new XML compiler option can be used to specify that the tags in the output of the XMLCHAR built-in function be either in all upper case or in the case in which they were declared.
- For compilations that produce no messages, the compiler will now include a line saying "no compiler messages" where the compiler messages would have been listed.
- The MACRO preprocessor will support a new suboption that will allow the user to specify whether it should process only %INCLUDE statements or whether it should process all macro statements.
- The integrated SQL preprocessor will now generate DB2 precompiler style declares for all *LOB_FILE, *LOCATOR, ROWID, BINARY and VARBINARY SQL types, in addition to the BLOB, CLOB and DBCLOB SQL types already supported, when the LOB(DB2) SQL preprocessor option is selected.

Enhancements from V3R6

This release provides the following functional enhancements described in this and the other IBM PL/I books.

DB2 V9 support

- Support for STDSQL(YES/NO)
- Support for CREATE TRIGGER (aka multiple SQL statements)
- Support for FETCH CONTINUE
- Support for SQL style comments ('--') embedded in SQL statements
- Support for additional SQL TYPES including
  - SQL TYPE IS BLOB_FILE
  - SQL TYPE IS CLOB_FILE
  - SQL TYPE IS DBCLOB_FILE
  - SQL TYPE IS XML AS
  - SQL TYPE IS BIGINT
  - SQL TYPE IS BINARY
  - SQL TYPE IS VARBINARY
- The SQL preprocessor will also now list the DB2 coprocessor options

Debugging improvements

- Under TEST(NOSEPNAME), the name of the debug side file will not be saved in the object deck.

Performance improvements

- Support, under ARCH(7), of the z/OS extended-immediate facility
- Exploitation of the CLCLU, MVCLU, PKA, TP and UNPKA instructions under ARCH(6)
- Exploitation of the CVBG and CVDG instructions under ARCH(5)
- Expanded use of CLCLE
- Conversions involving DB2 date-time patterns are now inlined
- The ALLOCATION built-in function is now inlined
- Conversions with a drifting $ have been inlined
- Conditional code has been eliminated from assignments to PIC'(n)Z'

- Conversions from FIXED BIN to a PICTURE that specifies a scale factor have been inlined
- Assignments to BIT variables that have inherited dimensions but which have strides that are divisible by 8 will now be inlined

Usability enhancements

- The MAP output will now also include a list in order of storage offset (per block) of the AUTOMATIC storage used by the block
- Conformance checking has been extended to include structures
- Listings will now include 7 columns for the line number in a file
- The THREADID built-in function is now supported under z/OS
- The PICSPEC built-in function is now supported
- The new CEESTART option allows you to position the CEESTART csect at the start or end of the object deck
- The new PPCICS, PPMACRO and PPSQL options allow you to specify the default options to be used by the corresponding preprocessor
- The ENVIRONMENT option is now included in the ATTRIBUTES listing
- The DISPLAY option now supports a suboption to allow different DESC codes for DISPLAY with REPLY versus DISPLAY without REPLY
- The message flagging a semicolon in a comment will now include the line number of the line where the semicolon appears
- Flag assignments that can change a REFER object
- Flag use of KEYED DIRECT files without a KEY/KEYFROM clause
- Flag use of PICTURE as loop control variables

Enhancements from V3R5

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Debugging improvements

- Under TEST(SEPARATE), the vast majority of debug information will be written to a separate debug file
- AUTOMONITOR will include the target in assignments
- The AT ENTRY hook will now be placed after AUTOMATIC has been initialized, thereby eliminating the need to step into the block before looking at any variables

Performance improvements

- Generation of branch-relative instructions so that the need for base registers and transfer vectors will be significantly eliminated
- Support, under ARCH(6), of the z/OS long displacement facility
- Simple structures using REFER will be mapped inlined rather than via a library call
- For structures using REFER still mapped via a library call, less code will be generated if the REFER specifies the bound for an array of substructures
- Faster processing of duplicate INCLUDEs
- Conversions to PICTURE variables with an I or R in the last position will now be inlined (such conversions had already been inlined when the last character was a T)

- Conversions to PICTURE variables ending with one or more B's will now be inlined if the corresponding picture without the B's would have been inlined
- Conversions to from CHARACTER to PICTURE variables consisting only of X's will now be inlined

Usability enhancements

- All parts of the listing, etc will count the source file as file 0, the first include file as file 1, the second (unique) include file as file 2, etc
- Conformance checking extended to include arrays
- Listings will include the build dates for any preprocessors invoked
- One-byte FIXED BINARY arguments can be suppressed for easier ILC with COBOL
- Alternate DD names may be specified for SYSADATA, SYSXMLSD and SYSDEBUG
- RULES(NOLAXMARGINS) tolerates, under XNUMERIC, sequence numbers
- RULES(NOUNREF) flags unreferenced AUTOMATIC variables
- If an assignment to a variable is done via library call, the message flagging the library call will include the name of the target variable
- Flag one-time DO loops
- Flag labels used as arguments
- Flag ALLOCATE and FREE of non-PARAMETER CONTROLLED in FETCHABLE if PRV used
- Flag DEFINED and BASED larger than their base even if the base is declared later
- Flag implicit FIXED DEC to 8-byte integer conversions

Enhancements from V3R4

This release provides the following functional enhancements described in this and the other IBM PL/I books.

Migration enhancements

- Support sharing of CONTROLLED with old code
- Improve default initialization
- Ease decimal precision specification in ADD, DIVIDE and MULTIPLY
- Support old semantics for STRING of GRAPHIC
- Support old semantics for the DEFAULT statement
- Flag declares with storage overlay
- Lift restrictions on RETURN inside BEGIN
- Optionally flag semicolons in comments
- Support EXT STATIC initialized in assembler
- Flag invalid carriage control characters
- Flag more language misuse, especially with RETURN
- Support the REPLACEBY2 built-in function
- Optionally suppress FOFL on decimal assignments that would raise SIZE
- Flag more language handled differently than the old compiler

Performance improvements

- Improve code generated for INDEX and TRANSLATE
- Inline more assignments to pictures
- Improve the code generated for conversions of CHARACTER to PICTURE when the conversion would be done inline if the source were FIXED DEC
- Improve the code generated for packed decimal conversions
- Improve the code generated for some uses of REFER
- Inline compares of character strings of unknown length
- Reduce the amount of stack storage used for concatenates
- Inline more GET/PUT STRING EDIT statements
- Short-circuit more LE condition handling
- Inline more Or and And of BIN FIXED
- Inline SIGNED FIXED BIN(8) to ALIGNED BIT(8)
- Flag statements where the compiler generates a call to a library routine to map a structure at run time
- Lessen the amount of I/O used to produce the listing

Usability enhancements

- Optionally provide offsets in the AGGREGATE listing in hex
- Support DEC(31) only when needed via the LIMITS(FIXEDDEC(15,31)) option
- Allow comments in options
- Optionally flag FIXED DEC declares with even precision
- Optionally flag DEC to DEC assignments that could raise SIZE
- Flag DEC/PIC to PIC assignments that could raise SIZE
- Support LIKE without INIT via the NOINIT attribute
- Ease includes from PDS's under z/OS UNIX
- Support the LOWERCASE, MACNAME, TRIM and LOWERCASE built-in functions in the MACRO preprocessor
- Ease introduction of options via PTF
- Optionally disallow use of *PROCESS
- Optionally keep *PROCESS in MDECK
- Support one-time INCLUDE
- Support macro-determined INCLUDE name
- Support runtime string parameter checking
- Flag more possibly uninitialized variables
- Flag unusual compares that are likely to be coding errors
- The output of the STORAGE option is now formatted more nicely, and the output of the LIST option will now include the hex offset for each block from the start of the compilation unit.

Debugging improvements

- Better support for overlay hooks
- Easier resolution of CONTROLLED variables in LE dumps
- Always include user specified options in the listing

Enhancements from V3R3

This release also provides all of the functional enhancements offered in Enterprise PL/I V3R3, including the following:

More XML support

The XMLCHAR built-in function will write XML with the names and values of the elements of a referenced structure to a buffer and return the number of bytes written. This XML can then be passed to other applications, including code using the PL/I SAX parser, which want to consume it.

Improved performance

- The compilation time under OPT(2) will be significantly less than under Enterprise PL/I V3R2, especially for large programs.
- The compiler now uses the ED and EDMK instructions for inlined numeric conversions to PICTURE and CHARACTER. This results in faster, shorter code sequences and also in faster compilations.
- The compiler now generates better code for string comparisons. This also results in faster, shorter code sequences.
- The compiler now generates shorter, faster code for conversion from FIXED DECIMAL to PICTURE with trailing overpunch characters.
- The ARCH and TUNE compiler options now accept 5 as a valid sub-option. Under ARCH(5), the compiler will generate, when appropriate, some new z/Architecture® instructions such as NILL, NILH, OILL, OILH, LLILL, and LLILH.

Easier migration

- The new BIFPREC compiler option controls the precision of the FIXED BIN result returned by various built-in functions and thus provides for better compatibility with the OS PL/I compiler.
- The new BACKREG compiler option controls which register the compiler uses as the backchain register and thus allows for easier mixing of old and new object code.
- The new RESEXP compiler option controls the evaluation of restricted expressions in code, and thus provides for better compatibility with the OS PL/I compiler.
- The new BLKOFF compiler option provides for controlling the way offsets in the compiler's pseudo-assembler listing are calculated.
- The STORAGE compiler option causes the compiler to produce, as part of the listing, a summary, similar to that produced by the OS PL/I compiler, of the storage used by each procedure and begin-block.

Improved usability

- The new LAXDEF suboption of the RULES compiler option allows the use of so-called illegal defining without having the compiler generate E-level messages.
- The new FLOATINMATH compiler option offers easier control of the precision with which math functions are evaluated.
- The new MEMINDEX, MEMSEARCH(R) and MEMVERIFY(R) built-in functions provide the ability to search strings larger than 32K.
- The new ROUTCDE and DESC suboptions of the DISPLAY(WTO) compiler option offers control of the corresponding elements of the WTO.

- The compiler now stores in each object a short string that will be in storage even when the associated code runs and that records all the options used to produce that object. This allows various tools to produce better diagnostics.
- The compiler now issues messages identifying more of the places where statements have been merged or deleted.
- The PLIDUMP output now includes a hex dump of user static.
- The PLIDUMP output now includes the options used to compile each program in the Language Environment® traceback.
- The PLIDUMP output now includes more information on PL/I files.

Improved debug support

- BASED structures using REFER are now supported in DebugTool and in data-directed I/O statements (with the same restrictions as on all other BASED variables).
- BASED structures that are BASED on scalar members of other structures (which, in turn, may be BASED, etc) are now supported in DebugTool and in data-directed I/O statements (with the same restriction as on all other BASED variables).

Enhancements from V3R2

This release also provides all of the functional enhancements offered in Enterprise PL/I V3R2, including the following:

Improved performance

- The compiler now handles even more conversions by generating inline code which means these conversions will be done much faster than previously. Also, all conversions done by library call are now flagged by the compiler.
- The compiler-generated code now uses, in various situations, less stack storage.
- The compiler now generates much better code for references to the TRANSLATE built-in function.
- The compiler-generated code for SUBSCRIPTRANGE checking is now, for arrays with known bounds, twice as fast as before.
- The ARCH and TUNE options now support 4 as a suboption, thereby allowing exploitation of instructions new to the zSeries machines.
- ARCH(2), FLOAT(AFP) and TUNE(3) are now the default.

Easier migration

- Compiler defaults have been changed for easier migration and compatibility. The changed defaults are:
  - CSECT
  - CMPAT(V2)
  - LIMITS(EXTNAME(7))
  - NORENT
- The compiler now honors the NOMAP, NOMAPIN and NOMAP attributes for PROCs and ENTRYs with OPTIONS(COBOL).
- The compiler now supports PROCs with ENTRY statements that have differing RETURNS attribute in the same manner as did the old host compiler.
- The compiler will now assume OPTIONS(RETCODE) for PROCs and ENTRYs with OPTIONS(COBOL).
- The SIZE condition is no longer promoted to ERROR if unhandled.

- Various changes have been made to reduce compile time and storage requirements.
- The OFFSET option will now produce a statement offset table much like the ones it produced under the older PL/I compilers.
- The FLAG option now has exactly the same meaning as it had under the old compilers, while the new MAXMSG option lets you decide if the compiler should terminate after a specified number of messages of a given severity. For example, with FLAG(I) MAXMSG(E,10), you can now ask to see all I-level messages while terminating the compilation after 10 E-level messages.
- The AGGREGATE listing now includes structures with adjustable extents.
- The STMT option is now supported for some sections of the listing.
- The maximum value allowed for LINESIZE has been changed to 32759 for F-format files and to 32751 for V-format files.

Improved usability

- The defaults for compiler options may now be changed at installation.
- The integrated SQL preprocessor now supports DB2 Unicode.
- The compiler now generates information that allows Debug Tool to support Auto Monitor, whereby immediately before each statement is executed, all the values of all the variables used in the statement are displayed.
- The new NOWRITABLE compiler option lets you specify that even under NORENT and at the expense of optimal performance, the compiler should use no writable static when generating code to handle FILEs and CONTROLLED.
- The new USAGE compiler option gives you full control over the IBM or ANS behavior of the ROUND and UNSPEC built-in function without the other effects of the RULES(IBM|ANS) option.
- The new STDSYS compiler option lets you specify that the compiler should cause the SYSPRINT file to be equated to the C stdout file.
- The new COMPACT compiler option lets you direct the compiler to favor those optimizations which tend to limit the growth of the code.
- The LRECL for SYSPRINT has been changed to 137 to match that of the C/C++ compiler.
- POINTERs are now allowed in PUT LIST and PUT EDIT statements: the 8-byte hex value will be output.
- If specified on a STATIC variable, the ABNORMAL attribute will cause that variable to be retained even if unused.

Enhancements from V3R1

This release also provides all of the functional enhancements offered in Enterprise PL/I V3R1, including the following:

- Support for multithreading on z/OS
- Support for IEEE floating-point on z/OS
- Support for the ANSWER statement in the macro prepreprocessor
- SAX-style XML parsing via the PLISAXA and PLISAXB built-in subroutines
- Additional built-in functions:
  - CS
  - CDS
  - ISMAIN
  - LOWERCASE
  - UPPERCASE

Enhancements from VisualAge PL/I

This release also provides all of the functional enhancements offered in VisualAge® PL/I V2R2, including the following:

- Initial UTF-16 support via the WIDECHAR attribute

  There is currently no support yet for
  - WIDECHAR characters in source files
  - W string constants
  - use of WIDECHAR expressions in stream I/O
  - implicit conversion to/from WIDECHAR in record I/O
  - implicit endianness flags in record I/O

    If you create a WIDECHAR file, you should write the endianness flag ('fe_ff'wx) as the first two bytes of the file.
- DESCRIPTORS and VALUE options supported in DEFAULT statements
- PUT DATA enhancements
  - POINTER, OFFSET and other non-computational variables supported
  - Type-3 DO specifications allowed
  - Subscripts allowed
- DEFINE statement enhancements
  - Unspecified structure definitions
  - CAST and RESPEC type functions
- Additional built-in functions:
  - CHARVAL
  - ISIGNED
  - IUNSIGNED
  - ONWCHAR
  - ONWSOURCE
  - WCHAR
  - WCHARVAL
  - WHIGH
  - WIDECHAR
  - WLOW
- Preprocessor enhancements
  - Support for arrays in preprocessor procedures
  - WHILE, UNTIL and LOOP keywords supported in %DO statements
  - %ITERATE statement supported
  - %LEAVE statement supported
  - %REPLACE statement supported
  - %SELECT statement supported
  - Additional built-in functions:
    - COLLATE
    - COMMENT
    - COMPILEDATE
    - COMPILETIME
    - COPY
    - COUNTER
    - DIMENSION
    - HBOUND
    - INDEX
    - LBOUND
    - LENGTH

- MACCOL
- MACLMAR
- MACRMAR
- MAX
- MIN
- PARMSET
- QUOTE
- REPEAT
- SUBSTR
- SYSPARM
- SYSTEM
- SYSVERSION
- TRANSLATE
- VERIFY

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information.

If you have comments about this document or any other PL/I documentation, contact us in one of these ways:

- Use the Online Readers' Comment Form at

  `www.ibm.com/software/awdtools/rcf/`

  Or send an email to `comments@us.ibm.com`

  Ensure to include the name of the document, the publication number of the document, the version of PL/I, and, if applicable, the specific location (for example, page number) of the text that you are commenting on.
- Fill out the Readers' Comment Form at the back of this document, and return it by mail or give it to an IBM representative. If the form has been removed, address your comments to:

  International Business Machines Corporation
  Reader Comments
  H150/090
  555 Bailey Avenue
  San Jose, CA 95141-1003
  USA
- Fax your comments to this U.S. number: (800)426-7773.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Accessibility

Accessibility features assist users who have a disability, such as restricted mobility or limited vision, to use information technology content successfully. The accessibility features in z/OSAIX provide accessibility for Enterprise PL/IEnterprise PL/I for AIX.

Accessibility features

z/OSAIX includes the following major accessibility features:

- Interfaces that are commonly used by screen readers and screen-magnifier software
- Keyboard-only navigation
- Ability to customize display attributes such as color, contrast, and font size
- An accessible command-line interface through a direct attached or networked remote Windows or Linux system using AT and terminal emulation software

z/OS uses the latest W3C Standard, WAI-ARIA 1.0 (http://www.w3.org/TR/wai-aria/), to ensure compliance to US Section 508 (http://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-section-508-standards/section-508-standards) and Web Content Accessibility Guidelines (WCAG) 2.0 (http://www.w3.org/TR/WCAG20/). To take advantage of accessibility features, use the latest release of your screen reader in combination with the latest web browser that is supported by this product.

The Enterprise PL/I online product documentation in IBM Knowledge Center is enabled for accessibility. The accessibility features of IBM Knowledge Center are described at http://www.ibm.com/support/knowledgecenter/en/about/releasenotes.html.

Keyboard navigation

Users can access z/OS user interfaces by using TSO/E or ISPF.

Users can also access z/OS services by using IBM Rational® Developer for System z®.

For information about accessing these interfaces, see the following publications:

- *z/OS TSO/E Primer* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4p120)
- *z/OS TSO/E User's Guide* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ikj4c240/APPENDIX1.3)
- *z/OS ISPF User's Guide Volume I* (http://publib.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/ispzug70)
- IBM Rational Developer for System z Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSQ2R2/rdz_welcome.html?lang=en)

These guides describe how to use TSO/E and ISPF, 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.

For information about accessing AIX interfaces, see Accessibility features for AIX in the IBM System p and AIX Information Center.

Interface information

The Enterprise PL/I online product documentation is available in IBM Knowledge Center, which is viewable from a standard web browser.

PDF files have limited accessibility support. With PDF documentation, you can use optional font enlargement, high-contrast display settings, and can navigate by keyboard alone.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains the period or comma PICTURE symbols, you must set the screen reader to speak all punctuation.

Assistive technology products work with the user interfaces that are found in AIXz/OS. For specific guidance information, see the documentation for the assistive technology product that you use to accessAIXz/OS interfaces.

Related accessibility information

In addition to standard IBM help desk and support websites, IBM has established a TTY telephone service for use by deaf or hard of hearing customers to access sales and support services:

TTY service
800-IBM-3383 (800-426-3383)
(within North America)

IBM and accessibility

For more information about the commitment that IBM has to accessibility, see IBM Accessibility (www.ibm.com/able).

Part 1. Compiling your program

© Copyright IBM Corp. 1999, 2017

Chapter 1. Using compiler options and facilities

This chapter describes the options that you can use for the compiler, along with their abbreviations and IBM-supplied defaults.

Important: PL/I requires access to the Language Environment run time when you compile your applications.

With Enterprise PL/I, you can develop both 31-bit and 64-bit applications. When you compile code, use the LP(32) option for 31-bit applications, or the LP(64) option for 64-bit applications. However, the compiler behavior under LP(64) is different from that under LP(32). For more information, see Chapter 7, "Considerations for developing 64-bit applications," on page 195 and Chapter 6, "Link-editing and running for 64-bit programs," on page 187.

You can override most defaults when you compile your PL/I program. You can also override the defaults when you install the compiler.

All the compiler option default settings for a release is listed in the member with the name IBMXO*vrm*, where *v* is the version number, *r* is the release number and *m* is the modification number which is usually 0 (for example, IBMXO510 for V5R1 and IBMXO450 for V4R5), in the SIBMZSAM data set. The data set contains IBMXO*vrm* members for all supported PL/I releases. To see what has changed in the default settings and what new options are added from one release to the next, you can compare the IBMXO*vrm* files for the two releases. You can also use the IBMXO*vrm* files as templates for creating your own options files with your preferred settings.

Compile-time option descriptions

Most compiler options have a positive and negative form. The negative form is the positive with `NO` added at the beginning (as in TEST and NOTEST). Some options have only a positive form (as in SYSTEM).

There are three types of compiler options:

1. Simple pairs of keywords: a positive form that requests a facility, and an alternative negative form that inhibits that facility (for example, NEST and NONEST)
2. Keywords that allow you to provide a value list that qualifies the option (for example, FLAG(W))
3. A combination of 1 and 2 above (for example, NOCOMPILE(E))

Table 3 on page 4 lists all the compiler options with abbreviations (if any) and IBM-supplied default values. If an option has any suboptions that can be abbreviated, those abbreviations are described in the full description of the option.

For the sake of brevity, some of the options are described loosely in the table (for example, only one suboption of LANGLVL is mandatory, and similarly, if you specify one suboption of TEST, you do not have to specify the other). The full and completely accurate syntax is described in the topics that follow.

© Copyright IBM Corp. 1999, 2017

Table 3. Compile-time options, abbreviations, and IBM-supplied defaults

Compile-time option	Abbreviated name	z/OS default
AGGREGATE[(DEC|HEX)] | NOAGGREGATE	AG | NAG	NOAGGREGATE
ARCH(n)	-	ARCH(8)
ATTRIBUTES[(FULL|SHORT)] | NOATTRIBUTES	A | NA	NA [(FULL)][1]
BACKREG(5 | 11)	-	BACKREG(5)
BIFPREC(15 | 31)	-	BIFPREC(15)
BLANK('c')	-	BLANK('t')[2]
BLKOFF | NOBLKOFF	-	BLKOFF
BRACKETS('*symbol_1symbol_2*')	-	BRACKETS('[]')
CASERULES(KEYWORD(LOWER | MIXED | START | UPPER))	-	CASERULES(KEYWORD (MIXED))
CEESTART(FIRST | LAST)	-	CEESTART(FIRST)
CHECK(STORAGE | NOSTORAGE, CONFORMANCE | NOCONFORMANCE)	-	CHECK(NSTG, NOCONFORMANCE)
CMPAT(V2 | V3)	CMP	CMPAT(V2)
CODEPAGE(n)	CP	CODEPAGE(1140)
COMMON | NOCOMMON	-	NOCOMMON
COMPILE | NOCOMPILE[(W | E | S)]	C | NC	NOCOMPILE(S)
COPYRIGHT('string') | NOCOPYRIGHT	-	NOCOPYRIGHT
CSECT | NOCSECT	CSE | NOCSE	CSECT
CSECTCUT(n)	-	CSECTCUT(4)
CURRENCY('c')	CURR	CURRENCY($)
DBCS | NODBCS	-	NODBCS
DD(ddname-list)	-	DD(SYSPRINT,SYSIN, SYSLIB,SYSPUNCH, SYSLIN,SYSADATA, SYSXMLSD,SYSDEBUG)
DDSQL(ddname)	-	DDSQL('')
DECIMAL(FOFLONASGN | NOFOFLONASGN, FOFLONMULT | NOFOFLONMULT, FORCEDSIGN | NOFORCEDSIGN, TRUNCFLOAT | NOTRUNCFLOAT)	DEC	DEC(FOFLONASGN, NOFOFLONMULT, NOFORCEDSIGN, NOTRUNCFLOAT)
DECOMP | NODECOMP	-	NODECOMP
DEFAULT(*attribute* | *option*)	DFT	See DEFAULT
DEPRECATE(BUILTIN(*built-in-name*) | ENTRY(*entry-name*) | INCLUDE(*filename*) | STMT(*statement-name*) | VARIABLE(*variable-name*))	-	DEPRECATE(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())
DEPRECATENEXT(BUILTIN(*built-in-name*) | ENTRY(*entry-name*) | INCLUDE(*filename*) | STMT(*statement-name*) | VARIABLE(*variable-name*))	-	DEPRECATENEXT(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())
DISPLAY (STD | WTO(ROUTCDE(x) DESC(y) REPLY(z)))	-	DISPLAY(WTO)
DLLINIT | NODLLINIT	-	NODLLINIT

Table 3. Compile-time options, abbreviations, and IBM-supplied defaults (continued)

Compile-time option	Abbreviated name	z/OS default
EXIT | NOEXIT	-	NOEXIT
EXTRN(FULL | SHORT)	-	EXTRN(FULL)
EXPORTALL	-	EXPORTALL
FILEREF | NOFILEREF	-	FILEREF
FLAG[(I | W | E | S)]	F	FLAG(W)
FLOAT(DFP | NODFP)	-	FLOAT(NODFP)
FLOATINMATH(ASIS | LONG | EXTENDED)	-	FLOATINMATH(ASIS)
GOFF | NOGOFF	-	NOGOFF
GONUMBER(SEPARATE | NOSEPARATE) | NOGONUMBER	GN | NGN	NOGONUMBER
GRAPHIC | NOGRAPHIC	GR | NGR	NOGRAPHIC
IGNORE(ASSERT | DISPLAY | PUT) | NOIGNORE	-	NOIGNORE
HEADER(SOURCE | FILE)	-	SOURCE
INCAFTER([PROCESS(*filename*)])	-	INCAFTER()
INCDIR('*directory name*') | NOINCDIR	-	NOINCDIR
INCLUDE | NOINCLUDE	-	INCLUDE
INCPDS('*PDS name*') | NOINCPDS	-	NOINCPDS
INITAUTO([SHORT | FULL]) | NOINITAUTO	-	NOINITAUTO
INITBASED | NOINITBASED	-	NOINITBASED
INITCTL | NOINITCTL	-	NOINITCTL
INITSTATIC | NOINITSTATIC	-	NOINITSTATIC
INSOURCE[(FULL|SHORT)(ALL|FIRST)] | NOINSOURCE	IS | NIS	NOINSOURCE
INTERRUPT | NOINTERRUPT	INT | NINT	NOINTERRUPT
JSON(CASE(UPPER | ASIS))	-	JSON(CASE(UPPER))
LANGLVL(NOEXT | OS)	-	LANGLVL(OS)
LIMITS(*options*)	-	See "LIMITS" on page 45
LINECOUNT(*n*)	LC	LINECOUNT(60)
LINEDIR | NOLINEDIR	-	NOLINEDIR
LIST | NOLIST	-	NOLIST
LISTVIEW(SOURCE | AFTERMACRO | AFTERCICS | AFTERSQL | AFTERALL)	-	LISTVIEW(SOURCE)
LP(32 | 64)	-	LP(32)
MACRO | NOMACRO	M | NM	NOMACRO
MAP | NOMAP	-	NOMAP
MARGINI('c') | NOMARGINI	MI | NMI	NOMARGINI
MARGINS(m,n[,c])| NOMARGINS	MAR(m,n)	MARGINS F-format: (2,72) V-format: (10,100)
MAXBRANCH(max)	-	MAXBRANCH(2000)
MAXGEN(n)	-	MAXGEN(100000)
MAXMEM(n)	MAXM	MAXMEM(1048576)
MAXMSG(I | W | E | S,n)	-	MAXMSG(W,250)

Table 3. Compile-time options, abbreviations, and IBM-supplied defaults (continued)

Compile-time option	Abbreviated name	z/OS default
MAXNEST(BLOCK(x) DO(y) IF(z))	-	MAXNEST(BLOCK(17) DO(17) IF(17))
MAXSTMT(n)	-	MAXSTMT(4096)
MAXTEMP(n)	-	MAXTEMP(50000)
MDECK | NOMDECK	MD | NMD	NOMDECK
MSGSUMMARY[(XREF | NOXREF)] | NOMSGSUMMARY	-	NOMSGSUMMARY
NAME[('*external name*')] | NONAME	N	NONAME
NAMES('*lower*'[,*upper*])	-	NAMES('#@$','#@$')
NATLANG(ENU | UEN)	-	NATLANG(ENU)
NEST | NONEST	-	NONEST
NOT	-	NOT('¬')
NULLDATE | NONULLDATE	-	NONULLDATE
NUMBER | NONUMBER	NUM | NNUM	NUMBER
OBJECT | NOOBJECT	OBJ | NOBJ	OBJECT
OFFSET | NOOFFSET	OF | NOF	NOOFFSET
OFFSETSIZE(n)[3]	-	OFFSETSIZE(4)
ONSNAP(STRINGRANGE, STRINGSIZE) | NOONSNAP	-	NOONSNAP
OPTIMIZE(0 | 2 | 3) | NOOPTIMIZE	OPT | NOPT	OPT(0)
OPTIONS[(ALL|DOC)] | NOOPTIONS	OP | NOP	NOOPTIONS
OR('c')	-	OR(' | ')
PP(*pp-name*) | NOPP	-	NOPP
PPCICS('string') | NOPPCICS	-	NOPPCICS
PPINCLUDE('string') | NOPPINCLUDE	-	NOPPINCLUDE
PPLIST(KEEP | ERASE)	-	PPLIST(KEEP)
PPMACRO('string') | NOPPMACRO	-	NOPPMACRO
PPSQL('string') | NOPPSQL	-	NOPPSQL
PPTRACE | NOPPTRACE	-	NOPPTRACE
PREFIX(*condition*)	-	See PREFIX
PRECTYPE(ANS | DECDIGIT | DECRESULT)	-	PRECTYPE(ANS)
PROCEED | NOPROCEED[(W | E | S)]	PRO | NPRO	NOPROCEED(S)
PROCESS[(KEEP | DELETE)] | NOPROCESS	-	PROCESS(DELETE)
QUOTE("")	-	QUOTE("")
REDUCE | NOREDUCE	-	REDUCE
RENT | NORENT	-	NORENT
RESEXP | NORESEXP	-	RESEXP
RESPECT([DATE])	-	RESPECT()
RTCHECK(NULLPTR | NONULLPTR | NULL370)	-	RTCHECK(NONULLPTR)
RULES(*options*)	-	See "RULES" on page 70
SEMANTIC | NOSEMANTIC[(W | E | S)]	SEM | NSEM	NOSEMANTIC(S)
SERVICE('*service string*') | NOSERVICE	SERV | NOSERV	NOSERVICE

Table 3. Compile-time options, abbreviations, and IBM-supplied defaults (continued)

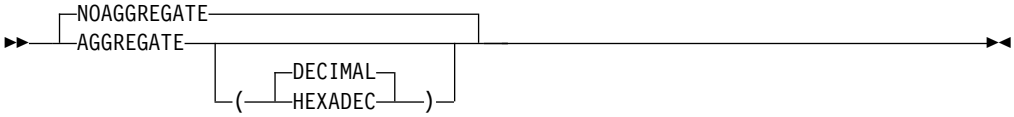
Compile-time option	Abbreviated name	z/OS default
SOURCE | NOSOURCE	S | NS	NOSOURCE
SPILL(n)	SP	SPILL(512)
STATIC(FULL | SHORT)	-	STATIC(SHORT)
STDSYS | NOSTDSYS	-	NOSTDSYS
STMT | NOSTMT	-	NOSTMT
STORAGE | NOSTORAGE	STG | NSTG	NOSTORAGE
STRINGOFGRAPHIC(CHAR | GRAPHIC)	-	STRINGOFGRAPHIC (GRAPHIC)
SYNTAX | NOSYNTAX[(W | E | S)]	SYN | NSYN	NOSYNTAX(S)
SYSPARM('string')	-	SYSPARM('')
SYSTEM(MVS | CICS | IMS | TSO | OS)	-	SYSTEM(MVS)
TERMINAL | NOTERMINAL	TERM | NTERM	
TEST(*options*) | NOTEST	-	See "TEST" on page 88[4]
UNROLL(AUTO | NO)		UNROLL(AUTO)
USAGE(*options*)	-	See "USAGE" on page 92
WIDECHAR(BIGENDIAN | LITTLEENDIAN)	WCHAR	WIDECHAR(BIGENDIAN)
WINDOW(w)	-	WINDOW(1950)
WRITABLE | NOWRITABLE[(FWS|PRV)]	-	WRITABLE
XINFO(*options*)	-	XINFO(NODEF,NOMSG, NOSYM,NOSYN,NOXML)
XML(CASE(UPPER | ASIS))	-	XML(CASE(UPPER))
XREF[(FULL | SHORT)(EXPLICIT | IMPLICIT)] | NOXREF	X | NX	NX [(FULL)][1]

Notes:
1. FULL is the default suboption if the suboption is omitted with ATTRIBUTES or XREF.
2. The default value for the BLANK character is the tab character with value '05'x.
3. The OFFSETSIZE option is ignored if the LP(32) option is in effect.
4. (ALL,SYM) is the default suboption if the suboption is omitted with TEST.

The following topics describe the options in alphabetical order. For those options specifying that the compiler is to list information, only a brief description is included. For details about the generated listing, see "Using the compiler listing" on page 102.

AGGREGATE

The AGGREGATE option creates an Aggregate Length Table that gives the lengths of arrays and major structures in the source program in the compiler listing.



ABBREVIATIONS: AG, NAG

The suboptions of the AGGREGATE option determine how the offsets of subelements are displayed in the Aggregate Length Table:

DECIMAL
All offsets are displayed in decimal.

HEXADEC
All offsets are displayed in hexadecimal.

In the Aggregate Length Table, the length of an undimensioned major or minor structure is always expressed in bytes, but the length might not be accurate if the major or minor structure contains unaligned bit elements.

The Aggregate Length Table includes structures but not arrays that have non-constant extents. However, the sizes and offsets of elements within structures with non-constant extents might be inaccurate or specified as *.

ARCH

The ARCH option specifies the architecture for which the instructions of executable programs are to be generated. It allows the optimizer to take advantage of specific hardware instruction sets.

You can specify the following values for the ARCH level:

8 Generates code that uses instructions available on the 2098-xxx models (IBM System z10® BC) and 2097-xxx models (IBM System z10 EC) in z/Architecture mode.

Specifically, these ARCH(8) machines and subsequent generations add instructions supported by the general instruction extensions facility, which can be exploited by the compiler.

9 Generates code that uses instructions available on the 2817-xxx (IBM zEnterprise 196 (z196)) and 2818-xxx models (IBM zEnterprise 114 (z114)) models in z/Architecture mode.

Specifically, these ARCH(9) machines and their follow-ons add instructions supported by the following facilities, which can be exploited by the compiler:

- The high-word facility
- The population count facility
- The distinct-operands facility
- The floating-point extension facility
- The load/store-on-condition facility

For further information about these facilities, see the *z/Architecture Principles of Operation*.

10 Generates code that uses instructions available on the 2827-xxx (IBM zEnterprise EC12) models in z/Architecture mode.

Specifically, these ARCH(10) machines and their follow-ons add instructions supported by the following facilities, which can be exploited by the compiler:

- The execution-hint facility
- The load-and-trap facility
- The miscellaneous-instructions-extension facility
- The transactional-execution facility

For further information about these facilities, see *z/Architecture Principles of Operation*.

11 Generates code that uses instructions available on the 2964-xxxx (IBM z13™) models in z/Architecture mode.

Specifically, these ARCH(11) machines and their follow-ons add instructions supported by the following facilities:

- Vector facility

Notes:

1. If you specify an ARCH value less than 8, the compiler resets it to 8.
2. The x in the model numbers above (such as 2084-xxx) is a wildcard and stands for any alphanumeric machine of that type.
3. Code that is compiled at ARCH(*n*) runs on machines in the ARCH(*m*) group if and only if *m* >= *n*.
4. You can mix code that is compiled with different ARCH levels without any restrictions.

ASSERT

The ASSERT option controls whether ASSERT statements call a default library routine that will raise the ASSERTION condition or a routine provided by the user.

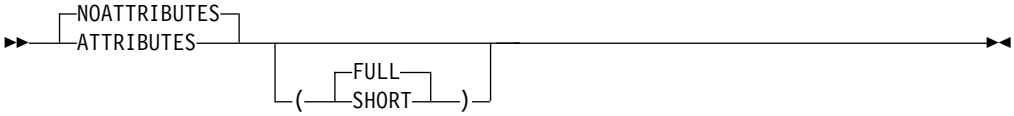
The default is ASSERT(ENTRY).

ENTRY
Specifies that ASSERT statements will call user-provided routines with the interfaces documented in the *PL/I Language Reference*. For details, see lsh-assert.dita.

CONDITION
Specifies that ASSERT statements will call library routines that will raise the ASSERTION condition with an appropriate ONCODE built-in function.

ATTRIBUTES

The ATTRIBUTES option specifies that the compiler includes a table of source-program identifiers and their attributes in the compiler listing.



ABBREVIATIONS: A, NA, F, S

FULL
All identifiers and attributes are included in the compiler listing. FULL is the default.

SHORT
Unreferenced identifiers are omitted, making the listing more manageable.

If you include both ATTRIBUTES and XREF (which creates a cross-reference table), the two tables are combined. However, if the SHORT and FULL suboptions are in conflict, the last option specified is used. For example, if you specify ATTRIBUTES(SHORT) XREF(FULL), FULL applies to the combined listing.

BACKREG

The BACKREG option controls the backchain register, which is the register used to pass the address of the automatic storage of a parent routine when a nested routine is invoked.

Note: Under the LP(64) option, the BACKREG option is ignored.

For best compatibility with PL/I for MVS & VM, OS PL/I V2R3, and earlier compilers, use BACKREG(5).

All routines that share an ENTRY VARIABLE must be compiled with the same BACKREG option, and it is strongly recommended that all code in application be compiled with the same BACKREG option.

Note that code compiled with VisualAge PL/I for OS/390® effectively uses the BACKREG(11) option. Code compiled with Enterprise PL/I V3R1 or V3R2 also uses the BACKREG(11) option by default.

BIFPREC

The BIFPREC option controls the precision of the FIXED BIN result returned by various built-in functions.

For best compatibility with PL/I for MVS & VM, OS PL/I V2R3, and earlier compilers, use BIFPREC(15).

BIFPREC affects the following built-in functions:

- COUNT
- INDEX
- LENGTH
- LINENO
- ONCOUNT
- PAGENO
- SEARCH

- SEARCHR
- SIGN
- VERIFY
- VERIFYR

The effect of the BIFPREC compiler option is most visible when the result of one of the above built-in functions is passed to an external function that has been declared without a parameter list. For example, consider the following code fragment:

```
dcl parm char(40) var;
dcl funky ext entry( pointer, fixed bin(15) );
dcl beans ext entry;
call beans( addr(parm), verify(parm), ' ' );
```

Suppose that the function `beans` actually declares its parameters as POINTER and FIXED BIN(15). If the preceding code is compiled with the option BIFPREC(31) and if it is run on a big-endian system such as z/OS, the compiler will pass a 4-byte integer as the second argument and the second parameter will be zero.

Note that the function `funky` will work on all systems with either option.

The BIFPREC option does not affect the built-in functions DIM, HBOUND and LBOUND. The CMPAT option determines the precision of the FIXED BIN result returned by these three functions:

- Under CMPAT(V1), these array-handling functions return a FIXED BIN(15) result.
- Under CMPAT(V2) and CMPAT(LE), they return a FIXED BIN(31) result.
- Under CMPAT(V3), they return a FIXED BIN(63) result.

BLANK

The BLANK option specifies up to ten alternate symbols for the blank character.

Note: Do not code any blanks between the quotation marks.

The IBM-supplied default code point for the BLANK symbol is '05'X.

char
A single SBCS character

You cannot specify any of the alphabetic characters, digits, or special characters defined in the *PL/I Language Reference*.

If you specify the BLANK option, the standard blank symbol is still recognized as a blank.

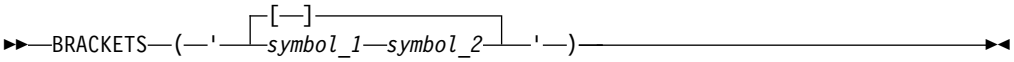
BLKOFF

The BLKOFF option controls whether the offsets shown in the pseudo-assembler listing (produced by the LIST option) and the statement offset listing (produced by the OFFSET option) are from the start of the current module or from the start of the current procedure.

The pseudo-assembler listing also includes the offset of each block from the start of the current module (so that the offsets shown for each statement can be translated to either block or module offsets).

BRACKETS

The BRACKETS option specifies the symbols that the SQL preprocessor accepts as the left and right brackets in SQL array references.



symbol_1
Specifies the symbol recognized as the left bracket in SQL array references.

symbol_2
Specifies the symbol recognized as the right bracket in SQL array references.

Note: The two values specified must be different from each other and must not be characters used in the PL/I character set or in other PL/I options such as NAMES, NOT, or OR.

The default is BRACKETS('[]').

Related information:

"NAMES" on page 56
The NAMES option specifies the *extralingual characters* that are allowed in identifiers.

"NOT" on page 56
The NOT option specifies up to seven alternate symbols that can be used as the logical NOT operator.

"OR" on page 61
The OR option specifies up to seven alternate symbols as the logical OR operator. These symbols are also used as the concatenation operator, which is defined as two consecutive logical OR symbols.

CASE

The CASE option controls whether some names will be shown in uppercase or in the same format as they appear in the source program.

The default is CASE(UPPER).

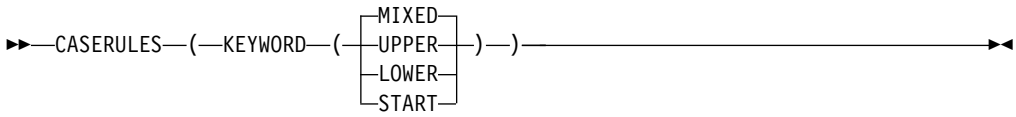
UPPER
Specifies that all names will be shown in uppercase.

ASIS
Specifies that names will be shown in the case in which they appear in the source in

- The AGGREGATE, ATTRIBUTES, and XREF listings.
- The values returned by the PACKAGENAME, PROCNAME and ORDINALNAME built-in functions.
- PUT DATA output as long as the TEST option is not used and as long as the compilation unit contains no GET DATA statements.

CASERULES

The CASERULES option controls the enforcement of case rules for keywords.



LOWER
Instructs the compiler to flag any keyword that is not in lowercase.

MIXED
Instructs the compiler to accept all keywords as they are coded. MIXED is the default.

START
Instructs the compiler to flag any keyword whose first letter is not in uppercase or whose remaining letters are not in lowercase.

UPPER
Instructs the compiler to flag any keyword that is not in uppercase.

Notes:

1. The CASERULES option does not apply to elements of the OPTIONS and ENVIRONMENT attributes.
2. The CASERULES option does not apply to any of the preprocessors.

CEESTART

The CEESTART option specifies whether the compiler should place the CEESTART csect before or after all the other generated object code.

Note: Under the LP(64) option, the CEESTART option is ignored.

Under the CEESTART(FIRST) option, the compiler places the CEESTART csect before all the other generated object code; however, under the CEESTART(LAST) option, the compiler places it after all the other generated object code.

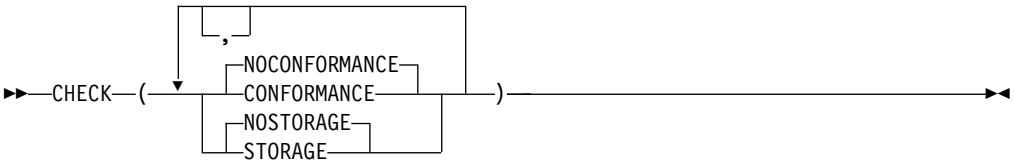
Using CEESTART(FIRST) will cause the binder to choose CEESTART as the entry point for a module if no ENTRY card is specified during the bind step.

If you want to use linker CHANGE cards, you must use the CEESTART(LAST) option.

However, MAIN routines should be linked with an ENTRY CEESTART linkage editor card. But, if you use the CEESTART(LAST) option, you must include an ENTRY CEESTART card when linking your MAIN routine.

CHECK

The CHECK option specifies whether the compiler should generate special code to detect various programming errors.



ABBREVIATIONS: STG, NSTG

CONFORMANCE | NOCONFORMANCE

Specifying CHECK(CONFORMANCE) causes the compiler to generate, under the following circumstances, code that checks at run time if the attributes of the arguments passed to a procedure match those of the declared parameters:

- If a parameter is a string (or an array of strings) declared with a constant length, then the STRINGSIZE condition will be raised if the argument passed does not have matching length.
- If a parameter is a string (or an array of strings), then the STRINGSIZE condition will be raised if the argument does not have the same length type (VARYING, NONVARYING or VARYINGZ).
- If a parameter is an array (of scalars or structures), then the SUBSCRIPTRANGE condition will be raised if any constant bounds do not match those of the passed argument. The SUBSCRIPTRANGE condition will also be raised if all the extents are constant and the size and spacing of the array elements in the argument do not match those in the parameter. Arrays inside a structure are not checked.
- If a parameter is a structure or union with constant extents, then the SUBSCRIPTRANGE condition will be raised if the offset of the last element does not match that of the passed argument.
- If the procedure has the RETURNS BYADDR attribute and that attribute specifies a string type, then the STRINGSIZE condition will be raised if the string passed for the RETURNS value does not have matching length.

This extra code will not be generated if any of the following conditions are true:

- The NODESCRIPTOR option applies to the procedure.
- The block contains ENTRY statements.
- The CMPAT(LE) option is in effect.

STORAGE | NOSTORAGE

When you specify CHECK(STORAGE), the compiler calls slightly different library routines for ALLOCATE and FREE statements (except when these statements occur within an AREA).

Note: The STORAGE suboption is not supported under the LP(64) option.

The following built-in functions, described in the *PL/I Language Reference*, can be used only when CHECK(STORAGE) has been specified:

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

AMODE(24) is not recommended for Enterprise PL/I applications. For code compiled with the CHECK(STORAGE) option, if you have to use AMODE(24), you must also specify the HEAP(,,BELOW) runtime option.

CMPAT

The CMPAT option specifies whether object compatibility with OS PL/I Version 1, OS PL/I Version 2, PL/I for MVS & VM, or Enterprise PL/I for z/OS is to be maintained for programs sharing strings, AREAs, arrays, or structures.

Note: Under the LP(64) option, the CMPAT option is ignored; effectively, CMPAT(V3) is always on.

ABBREVIATIONS: CMP

LE Under CMPAT(LE), your program can share strings, AREAs, arrays, or structures only with programs compiled with VisualAge PL/I for OS/390 or Enterprise PL/I for z/OS and only as long as the CMPAT(V1) and CMPAT(V2) options were not used when they were compiled. DB2 stored procedures must not be compiled with CMPAT(LE).

V1 Under CMPAT(V1), you can share strings, AREAs, arrays, or structures with programs compiled with the OS PL/I compiler and with programs compiled with later PL/I compilers as long as the CMPAT(V1) option was used. The JSON built-in functions are not supported under CMPAT(V1).

V2 Under CMPAT(V2), you can share strings, AREAs, arrays, or structures with programs compiled with the OS PL/I compiler and with programs compiled with later PL/I compilers as long as the CMPAT(V2) option was used.

V3 Under CMPAT(V3), you can share strings with programs compiled with the OS PL/I compiler and with programs compiled with later PL/I compilers as long as one of the CMPAT(V*) options was used. However, you cannot share AREAs, arrays, or structures with any code that was not compiled with CMPAT(V3).

All the modules in an application must be compiled with the same CMPAT option.

Mixing old and new code still has some restrictions. For information about these restrictions, see the *Enterprise PL/I for z/OS Compiler and Run-Time Migration Guide*.

The DFT(DESCLIST) option conflicts with the CMPAT(V*) options. If it is specified with any CMPAT(V*) option, a message will be issued and the DFT(DESCLOCATOR) option will be assumed.

Under CMPAT(V3), arrays can be declared with any value that an 8-byte integer can assume. However, unless the LP(64) option is used, the total size of an array currently still has the same limit as an array declared under CMPAT(V2).

Under CMPAT(V3), the following built-in functions will always return a FIXED BIN(63) result:
- CURRENTSIZE/CSTG
- DIMENSION
- HBOUND
- LBOUND
- LOCATION
- SIZE/STG

Because these functions will return 8-byte integer values, under CMPAT(V3), the second option in the FIXEDBIN suboption of the LIMITS option must be 63.

However, even under CMPAT(V3), statement and format label constants must be specified using 4-byte integers.

CODEPAGE

The CODEPAGE option specifies the code page used for conversions between CHARACTER and WIDECHAR. The option also specifies the default code page used by the PLISAX built-in subroutines.

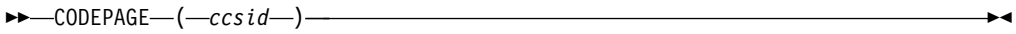


Table 4. Supported CCSIDs

01047	01145	00273	00297
01140	01146	00277	00500
01141	01147	00278	00871
01142	01148	00280	00819
01143	01149	00284	00813
01144	00037	00285	00920
01025	01155		

The default CCSID 1140 is an equivalent of CCSID 37 (EBCDIC Latin-1, USA) but includes the Euro symbol.

COMMON

The COMMON option directs the compiler to generate CM linkage records for EXTERNAL STATIC variables.

Note: Under the LP(64) option, the COMMON option is ignored.

Under the COMMON option, if the NORENT option applies, CM linkage records will be generated for EXTERNAL STATIC variables that are not RESERVED and that contain no INITIAL values. This matches what the OS PL/I compiler does.

Under the NOCOMMON option, SD records will be written as was true in earlier releases of Enterprise PL/I.

The COMMON option must not be used with the RENT option or with LIMITS(EXTNAME(*n*)) if *n* > 7.

COMPILE

The COMPILE option causes the compiler to stop compiling after all semantic checking of the source program if it produces a message of a specified severity during preprocessing or semantic checking.

Whether the compiler continues or not depends on the severity of the error detected, as specified by the NOCOMPILE option in the list below. The NOCOMPILE option specifies that processing stops unconditionally after semantic checking.

ABBREVIATIONS: C, NC

COMPILE
Generates code unless a severe error or an unrecoverable error is detected. This suboption is equivalent to NOCOMPILE(S).

NOCOMPILE
Compilation stops after semantic checking.

NOCOMPILE(W)
No code generation if a warning, an error, a severe error, or an unrecoverable error is detected.

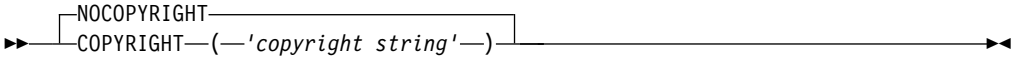
NOCOMPILE(E)
No code generation if an error, a severe error, or an unrecoverable error is detected.

NOCOMPILE(S)
No code generation if a severe error or an unrecoverable error is detected.

If the compilation is terminated by the NOCOMPILE option, the cross-reference listing and attribute listing can be produced; the other listings that follow the source program will not be produced.

COPYRIGHT

The COPYRIGHT option places a string in the object module, if generated. This string is loaded into memory with any load module into which this object is linked.



The string is limited to 1000 characters in length. However, if the string is longer than 100 characters, it will not be shown in the option listing.

To ensure that the string remains readable across locales, only characters from the invariant character set should be used.

CSECT

The CSECT option ensures that the object module, if generated, contains named CSECTs.

Use this option if you use SMP/E to service your product or to help debug your program.

ABBREVIATIONS: CSE, NOCSE

Under the NOCSECT option, the code and static sections of your object module are given default names.

Under the CSECT option, the code and static sections of your object module are given names that depend on the "package name", which is defined as follows:

- If the package statement was used, the "package name" is the leftmost label on the package statement.
- Otherwise, the "package name" is the leftmost label on the first procedure statement.

A "modified package name" of length 7 is then formed as follows:

- When the package name is less than 7 characters long, asterisks (*) are prefixed to it to make a modified package name that is 7 characters long.
- When the package name is more than 7 characters long, the first *n* and last *7 - n* characters are used to make the modified package name, where *n* is set by the CSECTCUT option.
- Otherwise, the package name is copied to the modified package name.

The code csect name is built by taking the modified package name and appending a 1 to it.

The static csect name is built by taking the modified package name and appending a 2 to it.

So, for a package named `SAMPLE`, the code csect name is `*SAMPLE1`, and the static csect name is `*SAMPLE2`.

CSECTCUT

The CSECTCUT option controls how the compiler, when processing the CSECT option, handles long names.

The CSECTCUT option has no effect unless you specify the CSECT option. It also has no effect if the "package name" used by the CSECT option has 7 or fewer characters.

The value *n* in the CSECTCUT option must be between 0 and 7.

If the "package name" used by the CSECT option has more than 7 characters, the compiler will collapse the name to 7 characters by taking the first *n* and last *7 - n* characters.

For example, consider a compilation consisting of one procedure with the name BEISPIEL:

- Under CSECTCUT(3), the compiler collapses the name to BEIPIEL.
- Under CSECTCUT(4), the compiler collapses the name to BEISIEL.

CURRENCY

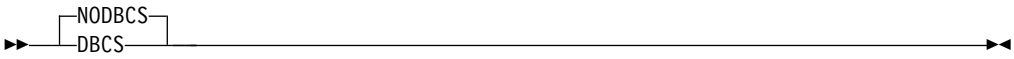
The CURRENCY option allows you to specify an alternate character to be used in picture strings instead of the dollar sign.

ABBREVIATIONS: CURR

x Character that you want the compiler and run time to recognize and accept as the dollar sign in picture strings

DBCS

The DBCS option ensures that the listing, if generated, is sensitive to the possible presence of DBCS even though the GRAPHIC option has not been specified.



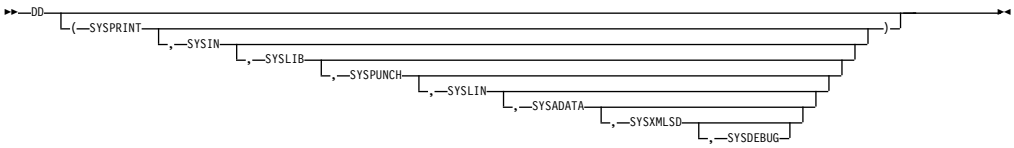
Under z/OS UNIX System Services, the NODBCS option causes the listing, if generated, to show all DBCS shift-codes as ".".

Under both batch and z/OS UNIX System Services, the NODBCS option ensures that the header text in the listing page does not contain unmatched shift codes.

The NODBCS option should not be specified if the GRAPHIC option is also specified.

DD

Using the DD option, you can specify alternate DD names for the various data sets used by the compiler.



Up to eight DD names can be specified. In order, they specify alternate DD names for the following data sets:

- SYSPRINT

- SYSIN
- SYSLIB
- SYSPUNCH
- SYSLIN
- SYSADATA
- SYSXMLSD
- SYSDEBUG

If you want to use ALTIN as the DD name for the primary compiler source file, you must specify DD(SYSPRINT,ALTIN). If you specify DD(ALTIN), SYSIN is used as the DDNAME for the primary compiler source file and ALTIN is used as the DD name for the compiler listing.

You can also use an asterisk (*) to indicate that the default DD name should be used. Thus DD(*,ALTIN) is equivalent to DD(SYSPRINT,ALTIN).

DDSQL

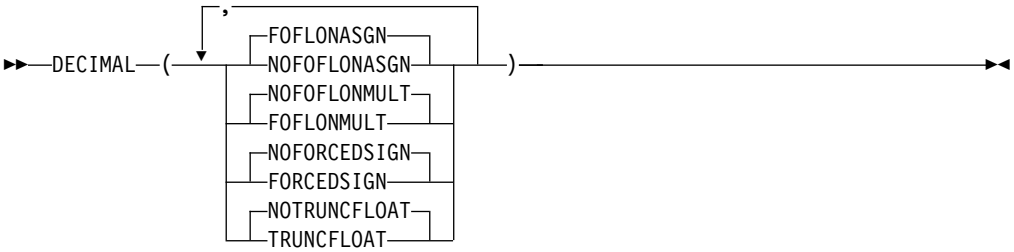
Using the DDSQL option, you can specify an alternate DD name for the data set that is used by the SQL preprocessor when the preprocessor resolves EXEC SQL INCLUDE statements.

Under the DDSQL('') option, the DD name that is used to resolve EXEC SQL INCLUDE statements is the DD name for SYSLIB from the DD compiler option.

This option might be useful when you are moving from the SQL precompiler to the integrated SQL preprocessor.

DECIMAL

The DECIMAL option specifies how the compiler should handle certain FIXED DECIMAL operations and assignments.



FOFLONASGN
When the FOFLONASGN option is enabled and the SIZE condition is disabled, the FOFLONASGN option requires the compiler to generate code that will raise the FIXEDOVERFLOW condition whenever a FIXED DECIMAL expression is assigned to a FIXED DECIMAL target and significant digits are lost.

Conversely, under the NOFOFLONASGN option, the compiler will generate code that will not raise the FIXEDOVERFLOW condition when significant digits are lost in such an assignment.

For example, given a variable A declared as FIXED DEC(5), the assignment A = A + 1 might raise FOFL under the FOFLONASGN option, but will never raise FOFL under the NOFOFLONASGN option.

Note, however, that under the NOFOFLONASGN option, the FIXEDOVERFLOW condition can still be raised by operations that produce a result with more digits than allowed by the FIXEDDEC suboption of the LIMITS option. For example, given a variable B declared as FIXED DEC(15) with the value 999_999_999_999_999 and given that the FIXEDDEC suboption of the LIMITS specifies the maximum precision as 15, then the assignment B = B + 1 will raise the FIXEDOVERFLOW condition (if FOFL is enabled).

FOFLONMULT

The FOFLONMULT option requires the compiler to generate code that will raise the FIXEDOVERFLOW condition for any use of the MULTIPLY built-in function that will produce a FIXED DEC result that is too large for the precision specified in the built-in function.

Conversely, under the NOFOFLONMULT option, the compiler will generate code that will produce a truncated result for any such use of the MULTIPLY built-in function.

Note that the use of the FOFLONMULT option changes the default language semantics (which is to truncate a too large result of the MULTIPLY built-in function applied to FIXED DEC - unless the SIZE condition is enabled).

FORCEDSIGN

The FORCEDSIGN option will force the compiler to generate extra code to ensure that whenever a FIXED DECIMAL result with the value zero is generated, the sign nibble of the result will have the value 'C'X. This option can cause the compiler to generate code that will perform much, much worse than the code generated under the NOFORCEDSIGN suboption.

Also, when this option is in effect, more data exceptions might occur when you run your code. For example, if you assign one FIXED DEC(5) variable to another FIXED DEC(5) variable, the compiler would normally generate an MVC instruction to perform the move. However, if the FORCEDSIGN option is in effect, to ensure that the result has the preferred sign, the compiler will generate a ZAP instruction to perform the move. If the source contains invalid packed decimal data, the ZAP instruction, but not the MVC instruction, will raise a decimal data exception.

Under this option, data exceptions might also be raised when one PICTURE variable is assigned to another PICTURE variable because that conversion usually involves an implicit conversion to FIXED DEC, which, under this option, will generate a ZAP instruction that will raise a data exception if the source contains invalid data.

Under DECIMAL(NOFORCEDSIGN), a "negative zero" might be produced by certain calculations. However, a programmer should rely on getting a negative zero only when assigning a negative literal to a FIXED DEC that cannot hold a value of that small a magnitude (such as assigning -.001 to a FIXED DEC(5,2)).

TRUNCFLOAT

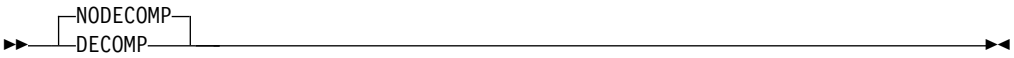
This suboption instructs the compiler how to handle assignments of float to fixed decimal when truncation might occur.

Under TRUNCFLOAT, if a hexadecimal float value is converted to FIXED DEC(*p*,*q*) where *p* <=18 and abs(*q*) <= *p*, and if the source value is too large for the target, the source value will be truncated and overflow will not be raised.

The default is NOTRUNCFLOAT.

DECOMP

The DECOMP option instructs the compiler to generate a listing section that gives a decomposition of expressions used in the compilation.



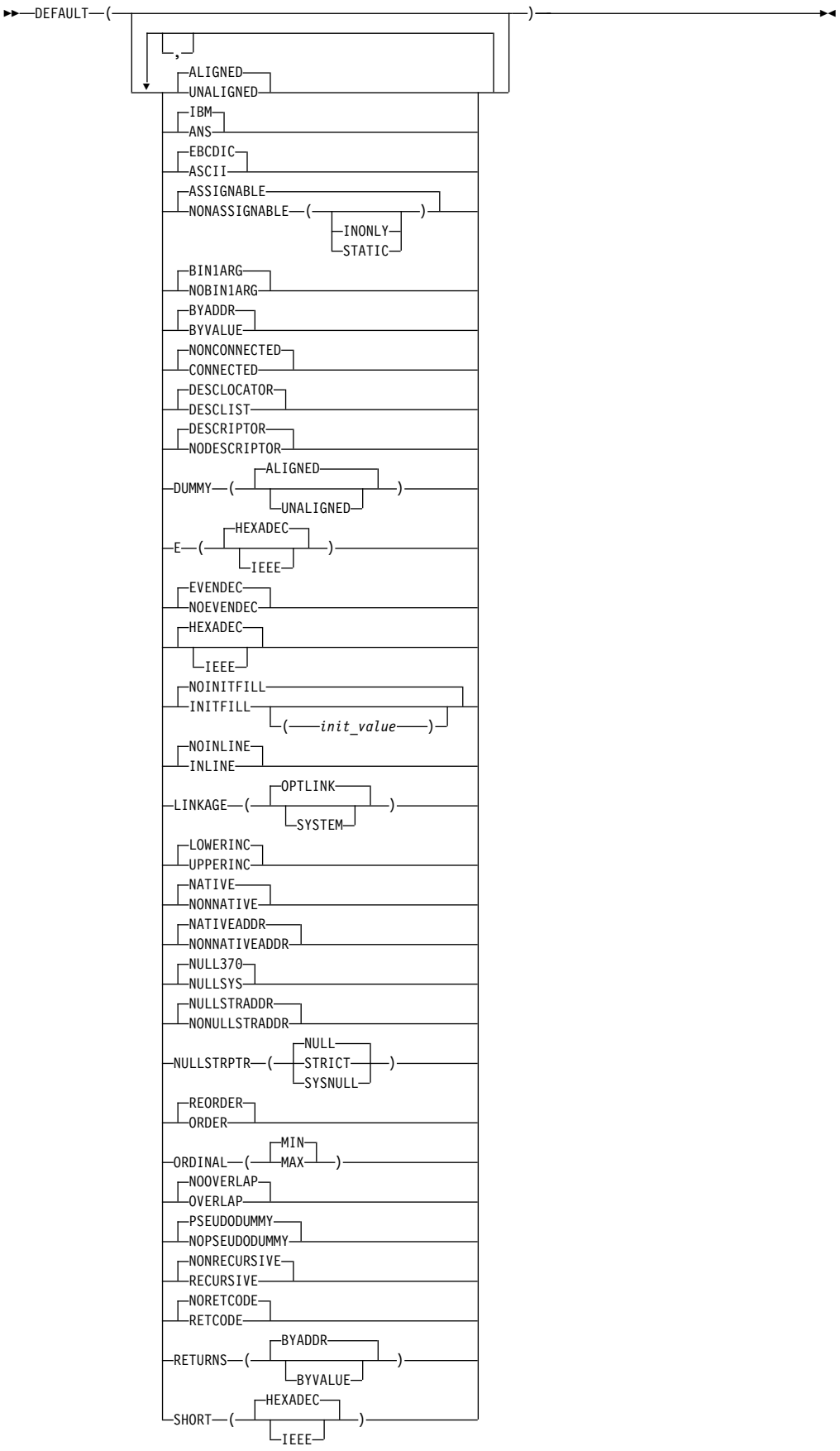
Under the DECOMP option, the compiler generates a listing section that shows all intermediate expressions and their attributes for all expressions used in the source program.

Under the NODECOMP option, the compiler does not generate this listing section.

The default is NODECOMP.

DEFAULT

The DEFAULT option specifies defaults for attributes and options. These defaults are applied only when the attributes or options are not specified or implied in the source code.


DEFAULT
(
,
ALIGNED
UNALIGNED
IBM
ANS
EBCDIC
ASCII
ASSIGNABLE
NONASSIGNABLE
(
INONLY
STATIC
)
BIN1ARG
NOBIN1ARG
BYADDR
BYVALUE
NONCONNECTED
CONNECTED
DESCLOCATOR
DESCLIST
DESCRIPTOR
NODESCRIPTOR
DUMMY
(
ALIGNED
UNALIGNED
)
E
(
HEXADEC
IEEE
)
EVENDEC
NOEVENDEC
HEXADEC
IEEE
NOINITFILL
INITFILL
(
init_value
)
NOINLINE
INLINE
LINKAGE
(
OPTLINK
SYSTEM
)
LOWERINC
UPPERINC
NATIVE
NONNATIVE
NATIVEADDR
NONNATIVEADDR
NULL370
NULLSYS
NULLSTRADDR
NONULLSTRADDR
NULLSTRPTR
(
NULL
STRICT
SYSNULL
)
REORDER
ORDER
ORDINAL
(
MIN
MAX
)
NOOVERLAP
OVERLAP
PSEUDODUMMY
NOPSEUDODUMMY
NONRECURSIVE
RECURSIVE
NORETCODE
RETCODE
RETURNS
(
BYADDR
BYVALUE
)
SHORT
(
HEXADEC
IEEE
)

ABBREVIATIONS: DFT, ASGN, NONASGN, NONCONN, CONN, INL, NOINL

ALIGNED | UNALIGNED

This suboption forces byte-alignment on all of your variables.

If you specify ALIGNED, all variables other than character, bit, graphic, and picture are given the ALIGNED attribute unless the UNALIGNED attribute is explicitly specified (possibly on a parent structure) or implied by a DEFAULT statement.

If you specify UNALIGNED, all variables are given the UNALIGNED attribute unless the ALIGNED attribute is explicitly specified (possibly on a parent structure) or implied by a DEFAULT statement.

ALIGNED is the default.

IBM | ANS

The suboption specifies whether to use IBM defaults or ANS SYSTEM defaults. The following table shows the arithmetic defaults for IBM and ANS.

Attributes	DEFAULT(IBM)	DEFAULT(ANS)
FIXED DECIMAL	(5,0)	(10,0)
FIXED BINARY	(15,0)	(31,0)
FLOAT DECIMAL	(6)	(6)
FLOAT BINARY	(21)	(21)

Under the IBM suboption, the default for variables with names beginning from I to N is FIXED BINARY, and the default for any other variables is FLOAT DECIMAL. If you select the ANS suboption, the default for all variables is FIXED BINARY.

IBM is the default.

ASCII | EBCDIC

This suboption sets the default for the character set used for the internal representation of character problem program data.

Specify ASCII only when compiling programs that depend on the ASCII character set collating sequence. Such a dependency exists, for example, if your program relies on the sorting sequence of digits or on lowercase and uppercase alphabetics. This dependency also exists in programs that create an uppercase alphabetic character by changing the state of the high-order bit.

Note: The compiler supports `A` and `E` as suffixes on character strings. The `A` suffix indicates that the string is meant to represent ASCII data, even if the EBCDIC compiler option is in effect. Alternately, the `E` suffix indicates that the string is EBCDIC, even when you select DEFAULT(ASCII).

```
'123'A is the same as '313233'X
'123'E is the same as 'F1F2F3'X
```

EBCDIC is the default.

ASSIGNABLE | NONASSIGNABLE

This option causes the compiler to apply the specified attribute to all static variables that are not declared with the ASSIGNABLE or NONASSIGNABLE attribute. The compiler flags statements in which NONASSIGNABLE variables are the targets of assignments.

ASSIGNABLE is the default.

INONLY
Specifying NONASSIGNABLE(INONLY) indicates that parameters declared with the INONLY attribute are given the NONASSIGNABLE attribute.

STATIC
Specifying NONASSIGNABLE(STATIC) indicates that STATIC variables are given the NONASSIGNABLE attribute.

The INONLY and STATIC suboptions have no effect on either variables with the ASSIGNABLE or NONASSIGNABLE attribute or structure members that inherit the ASSIGNABLE or NONASSIGNABLE attribute from a parent.

BYVALUE parameters are given the INONLY attribute after the resolution of the (NON)ASSIGNABLE attribute, and hence the NONASSIGNABLE(INONLY) suboption has no effect on BYVALUE parameters.

To specify the NONASSIGNABLE attribute to both STATIC and INONLY variables, you must specify the suboption NONASSIGNABLE(STATIC INONLY).

The NONASSIGNABLE attribute can be specified without any suboptions, in which case it means NONASSIGNABLE(STATIC).

BIN1ARG | NOBIN1ARG
This suboption controls how the compiler handles 1-byte REAL FIXED BIN arguments passed to an unprototyped function.

Under BIN1ARG, the compiler passes a FIXED BIN argument as is to an unprototyped function.

But under NOBIN1ARG, the compiler assigns any 1-byte REAL FIXED BIN argument passed to an unprototyped function to a 2-byte FIXED BIN temporary and pass that temporary instead.

Consider the following example:

```
dcl f1 ext entry;
dcl f2 ext entry( fixed bin(15) );

call f1( 1b );
call f2( 1b );
```

If you specify DEFAULT(BIN1ARG), the compiler passes the address of a 1-byte FIXED BIN(1) argument to the routine `f1` and the address of a 2-byte FIXED BIN(15) argument to the routine `f2`. However, if you specify DEFAULT(NOBIN1ARG), the compiler passes the address of a 2-byte FIXED BIN(15) argument to both routines.

Note that if the routine `f1` is a COBOL routine, passing a 1-byte integer argument to it might cause problems because COBOL has no support for 1-byte integers. In this case, using DEFAULT(NOBIN1ARG) might be helpful; but it might be better to specify the argument attributes in the entry declare statement.

BIN1ARG is the default.

BYADDR | BYVALUE
This suboption sets the default for whether arguments or parameters are passed by reference or by value. BYVALUE applies only to certain arguments and parameters. See the *PL/I Language Reference* for more information.

BYADDR is the default.

CONNECTED | NONCONNECTED
This suboption sets the default for whether parameters are connected or nonconnected. CONNECTED allows the parameter to be used as a target or source in record-oriented I/O or as a base in string overlay defining.

NONCONNECTED is the default.

DESCLIST | DESCLOCATOR
When you specify DEFAULT(DESCLIST), the compiler passes all descriptors in a list as a 'hidden' last parameter.

If you specify DEFAULT(DESCLOCATOR), parameters requiring descriptors are passed using a locator or descriptor in the same way as previous releases of PL/I. This allows old code to continue to work even if it passes a structure from one routine to a routine that is expecting to receive a pointer.

The DFT(DESCLIST) option conflicts with the CMPAT(V*) options, and if it is specified with any of them, a message will be issued and the DFT(DESCLOCATOR) option will be assumed.

DESCLOCATOR is the default.

DESCRIPTOR | NODESCRIPTOR
Using DESCRIPTOR with a PROCEDURE indicates that a descriptor list was passed, while DESCRIPTOR with ENTRY indicates that a descriptor list should be passed. NODESCRIPTOR results in more efficient code, but has the following restrictions:

- For PROCEDURE statements, NODESCRIPTOR is invalid if any of the parameters contains the following:
  - An asterisk (*) specified for the bound of an array, the length of a string, or the size of an area except if it is a VARYING or VARYINGZ string with the NONASSIGNABLE attribute
  - The NONCONNECTED attribute
  - The UNALIGNED BIT attribute
- For ENTRY declarations, NODESCRIPTOR is invalid if an asterisk (*) is specified for the bound of an array, the length of a string, or the size of an area in the ENTRY description list.

DESCRIPTOR is the default.

DUMMY(ALIGNED | UNALIGNED)
This suboption reduces the number of situations in which dummy arguments get created.

DUMMY(ALIGNED) indicates that a dummy argument should be created even if an argument differs from a parameter only in its alignment.
DUMMY(UNALIGNED) indicates that no dummy argument should be created for a scalar (except a nonvarying bit) or an array of such scalars if it differs from a parameter only in its alignment.

Consider the following example:

```
dcl
  1 a1 unaligned,
    2 b1   fixed bin(31),
    2 b2   fixed bin(15),
    2 b3   fixed bin(31),
    2 b4   fixed bin(15);

dcl x entry( fixed bin(31) );

call x( b3 );
```

If you specify DEFAULT(DUMMY(ALIGNED)), a dummy argument is created, while if you specify DEFAULT(DUMMY(UNALIGNED)), no dummy argument is created.

DUMMY(ALIGNED) is the default.

E (HEXADEC | IEEE)
The E suboption determines how many digits will be used for the exponent in E-format items.

If you specify E(IEEE), 4 digits will be used for the exponent in E-format items.

If you specify E(HEXADEC), 2 digits will be used for the exponent in E-format items.

If DFT(E(HEXADEC)) is specified, an attempt to use an expression whose exponent has an absolute value greater than 99 will cause the SIZE condition to be raised.

If the compiler option DFT(IEEE) is in effect, you should normally also use the option DFT(E(IEEE)). However, under this option, some E format items that would be valid under DFT(E(HEXADEC)) are not valid. For instance, under DFT(E(IEEE)), the statement `put skip edit(x) (e(15,8));` will be flagged because the E format item is invalid.

E(HEXADEC) is the default.

EVENDEC | NOEVENDEC
This suboption controls the compiler's tolerance of fixed decimal variables declared with an even precision.

Under NOEVENDEC, the precision for any fixed decimal variable is rounded up to the next highest odd number.

If you specify EVENDEC and then assign 123 to a FIXED DEC(2) variable, the SIZE condition is raised. If you specify NOEVENDEC, the SIZE condition is not raised.

EVENDEC is the default.

HEXADEC | IEEE
This suboption specifies the default representation that is used to hold all FLOAT variables and all floating-point intermediate results. This suboption also determines whether the compiler evaluates floating-point expressions using the hexadecimal or IEEE float instructions and math routines.

It is recommended that you use the IEEE option for programs that communicate with JAVA and also for programs that pass data to or receive data from platforms where IEEE is the default representation for floating-point data.

HEXADEC is the default.

INITFILL | NOINITFILL
This suboption controls the default initialization of automatic variables.

If you specify INITFILL with a hex value (nn), that value is used to initialize the storage that is used by all automatic variables in a block each time that block is entered. If you do not enter a hex value, the default is '00'.

Note that the hex value can be specified with or without quotation marks, but if it is specified with quotation marks, the string should not have an X suffix.

Under NOINITFILL, the storage that is used by an automatic variable can hold arbitrary bit patterns unless the variable is explicitly initialized.

INITFILL can cause programs to run slower and should not be specified in production programs. However, the INITFILL option produces code that runs faster than the LE STORAGE option. Also, during program development, this option is useful for detecting uninitialized automatic variables: a program that runs correctly with DFT(INITFILL('00')) and with DFT(INITFILL('ff')) probably has no uninitialized automatic variables.

NOINITFILL is the default.

INLINE | NOINLINE
This option sets the default for the inline procedure option.

Specifying INLINE allows your code to run faster but, in some cases, also creates a larger executable file. For more information about how inlining can improve the performance of your application, see Chapter 15, "Improving performance," on page 329.

NOINLINE is the default.

LINKAGE
Here is the linkage convention for procedure invocations:

OPTLINK
The default linkage convention for Enterprise PL/I. This linkage provides the best performance.

SYSTEM
The standard linking convention for system APIs.

Use LINKAGE(OPTLINK) for all routines called by or calling to JAVA and also for all routines called by or calling to C (unless the C code has been compiled with a nondefault linkage).

Use LINKAGE(SYSTEM) for all non-PL/I routines that expect the high-order bit to be on in the address of the last (and only the last) parameter.

Note that specifying OPTIONS(ASSEMBLER) for a PROCEDURE or an ENTRY forces LINKAGE(SYSTEM) regardless of the setting of this option.

LINKAGE(OPTLINK) is the default.

Note: The LINKAGE suboption is ignored under the LP(64) option.

LOWERINC | UPPERINC
If you specify LOWERINC, the compiler requires that the actual file names of INCLUDE files are in lowercase. If you specify UPPERINC, the compiler requires that the names are in uppercase.

Note: This suboption applies only to compilations under z/OS UNIX.

Under z/OS UNIX, the include name is built with the extension `.inc`. For example, under the DFT(LOWERINC) option, the statement `%INCLUDE STANDARD;` causes the compiler to try to include `standard.inc`. But, under the DFT(UPPERINC) option, the statement `%INCLUDE STANDARD;` causes the compiler to try to include `STANDARD.INC`.

LOWERINC is the default.

NATIVE | NONNATIVE
This suboption affects only the internal representation of fixed binary, ordinal, offset, area, and varying string data. When the NONNATIVE suboption is in effect, the NONNATIVE attribute is applied to all such variables not declared with the NATIVE attribute.

You should specify NONNATIVE only to compile programs that depend on the nonnative format for holding these kind of variables.

If your program bases fixed binary variables on pointer or offset variables (or conversely, pointer or offset variables on fixed binary variables), specify either of the following combinations of suboptions:
- Both the NATIVE and NATIVEADDR suboptions
- Both the NONNATIVE and NONNATIVEADDR suboptions.

Other combinations produce unpredictable results.

NATIVE is the default.

NATIVEADDR | NONNATIVEADDR

This suboption affects only the internal representation of pointers. When the NONNATIVEADDR suboption is in effect, the NONNATIVE attribute is applied to all pointer variables not declared with the NATIVE attribute.

If your program bases fixed binary variables on pointer or offset variables (or conversely, pointer or offset variables on fixed binary variables), specify either of the following combinations of suboptions:
- Both the NATIVE and NATIVEADDR suboptions
- Both the NONNATIVE and NONNATIVEADDR suboptions.

Other combinations produce unpredictable results.

NATIVEADDR is the default.

NULLSYS | NULL370

This suboption determines which value is returned by the NULL built-in function. If you specify NULLSYS, binvalue(null()) is equal to 0. If you want binvalue(null()) to equal 'ff_00_00_00'xn as is true with previous releases of PL/I, specify NULL370.

NULL370 is the default.

Note: The NULL370 or NULLSYS suboption is ignored under the LP(64) option.

NULLSTRADDR | NONULLSTRADDR

This suboption controls how the compiler handles null strings when null strings are passed as arguments.

Under NULLSTRADDR, when a null string is specified as an argument in an entry invocation, the compiler will pass the address of an initialized piece of automatic storage. This is compatible with what the OS PL/I and PL/I for MVS compilers did.

But under NONULLSTRADDR, when a null string is specified as an argument in an entry invocation, the compiler passes a null pointer as the address of the argument. This is compatible with what early releases of the Enterprise PL/I compiler did.

NULLSTRADDR is the default.

NULLSTRPTR

This suboption controls how the compiler handles null strings when null strings are assigned to POINTERs.

Under NULLSTRPTR(SYSNULL), the result of assigning '' to a POINTER is the same as assigning SYSNULL() to the pointer.

Under NULLSTRPTR(NULL), the result of assigning '' to a POINTER is the same as assigning NULL() to the pointer.

Under NULLSTRPTR(STRICT), assignments and comparisons of '' to POINTERs are flagged as invalid.

NULLSTRPTR(NULL) is the default.

ORDER | REORDER
This suboption affects the optimization of the object code. Specifying REORDER allows more optimization of your code. For detailed information, see Chapter 15, “Improving performance,” on page 329.

REORDER is the default.

ORDINAL(MIN | MAX)
If you specify ORDINAL(MAX), all ordinals whose definition does not include a PRECISION attribute is given the attribute PREC(31). Otherwise, they are given the smallest precision that covers their range of values.

ORDINAL(MIN) is the default.

OVERLAP | NOOVERLAP
If you specify OVERLAP, the compiler presumes the source and target in an assignment can overlap and generates, as needed, extra code in order to ensure that the result of the assignment is okay.

The OVERLAP suboption applies only to string variables. It has no effect on assignments of FIXED DECIMAL or other variable types. In those assignments, the source and target must not overlap.

NOOVERLAP produces code that performs better; however, if you use NOOVERLAP, you must ensure that the source and target never overlap.

NOOVERLAP is the default.

PSEUDODUMMY | NOPSEUDODUMMY
This suboption determines whether dummy arguments are created when a SUBSTR reference is specified as an argument to an unprototyped function.

If you specify PSEUDODUMMY, dummy arguments are created when a SUBSTR reference is specified as an argument to an unprototyped function.

If you specify NOPSEUDODUMMY, dummy arguments are not created when a SUBSTR reference is specified as an argument to an unprototyped function.

PSEUDODUMMY is the default.

RECURSIVE | NONRECURSIVE
When you specify DEFAULT(RECURSIVE), the compiler applies the RECURSIVE attribute to all procedures. If you specify DEFAULT(NONRECURSIVE), all procedures are nonrecursive except procedures with the RECURSIVE attribute.

NONRECURSIVE is the default.

RETCODE | NORETCODE
If you specify RETCODE, for any external procedure that does not have the RETURNS attribute, the compiler will generate extra code so that the procedure returns the integer value obtained by invoking the PLIRETV built-in function just before returning from that procedure.

If you specify NORETCODE, no special code is generated for procedures that do not have the RETURNS attribute.

NORETCODE is the default.

RETURNS (BYVALUE | BYADDR)
This suboption sets the default for how values are returned by functions. See the *PL/I Language Reference* for more information.

You must specify RETURNS(BYADDR) if your application contains ENTRY statements and the ENTRY statements or the containing procedure statement have the RETURNS option. You must also specify RETURNS(BYADDR) on the entry declarations for such entries.

RETURNS(BYADDR) is the default.

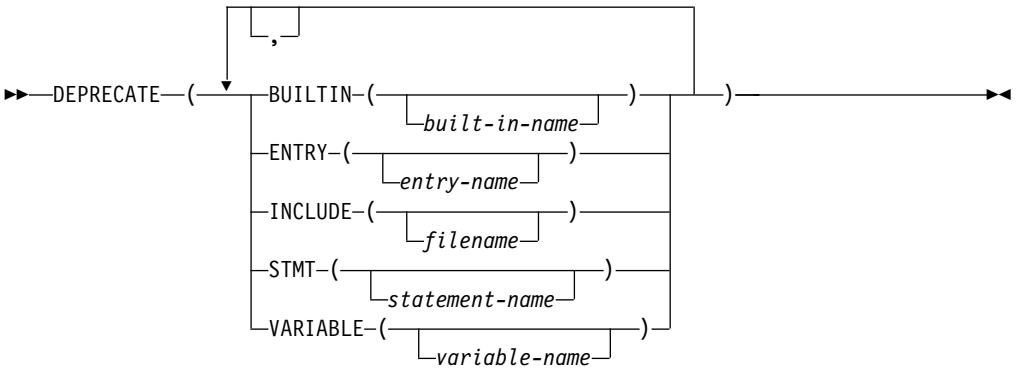
SHORT (HEXADEC | IEEE)
This suboption improves compatibility with other non-IBM UNIX compilers. SHORT (HEXADEC) maps FLOAT BIN (*p*) to a short (4-byte) floating point number if *p* <= 21. SHORT (IEEE) maps FLOAT BIN (*p*) to a short (4-byte) floating point number if *p* <= 24.

SHORT (HEXADEC) is the default.

Default: DEFAULT(ALIGNED IBM EBCDIC ASSIGNABLE BIN1ARG BYADDR NONCONNECTED DESCLOCATOR DESCRIPTOR DUMMY(ALIGNED) E(HEXADEC) EVENDEC HEXADEC NOINITFILL NOINLINE LINKAGE(OPTLINK) LOWERINC NATIVE NATIVEADDR NULL370 NULLSTRPTR(NULL) NULLSTRADDR REORDER ORDINAL(MIN) NOOVERLAP PSEUDODUMMY NONRECURSIVE NORETCODE RETURNS(BYADDR) SHORT(HEXADEC))

DEPRECATE

This option flags variable names, included file names, and statement names that you want to deprecate with error messages.



BUILTIN
Flags any declaration of *built-in-name* with the BUILTIN attribute.

built-in-name
Name of the BUILTIN variable

ENTRY
Flags any declaration of *entry-name* with the ENTRY attribute.

entry-name
Level-1 name

INCLUDE
Flags any %INCLUDE statement that includes *filename*.

filename
Name of the file

STMT
Flags all statements with the name as *statement-name*.

statement-name
Name of the statement

The names are identified by the initial keywords of PL/I statements. The STMT option accepts the following keywords:

allocate	assert	attach	begin	call	close	delay	delete
detach	display	exit	fetch	flush	free	get	goto
iterate	leave	locate	on	open	put	read	release
resignal	revert	rewrite	signal	stop	wait	write	

VARIABLE
Flags any declaration of *variable name* that does not have the BUILTIN or ENTRY attributes.

variable-name
Level-1 name

To specify the DEPRECATE option, you must specify at least one of these suboptions with a possible empty suboption list. For example, both of the following two specifications are invalid:

- DEPRECATE
- DEPRECATE(BUILTIN)

Specifying one of the suboptions does not change the setting of any of the other suboptions that are specified previously.

Specifying a suboption a second time replaces the previous specifications.

In all cases, there is no checking of the suboption lists.

Default: DEPRECATE(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())

Examples

- The following specifications are equivalent:

```
DEPRECATE(ENTRY(old)) DEPRECATE(BUILTIN(acos))
DEPRECATE(ENTRY(old) BUILTIN(acos))
```

- In the following example, `x` in the first specification is replaced by `y`:

```
DEPRECATE(BUILTIN(x)) DEPRECATE(BUILTIN(y))
DEPRECATE(BUILTIN(y))
```

DEPRECATENEXT

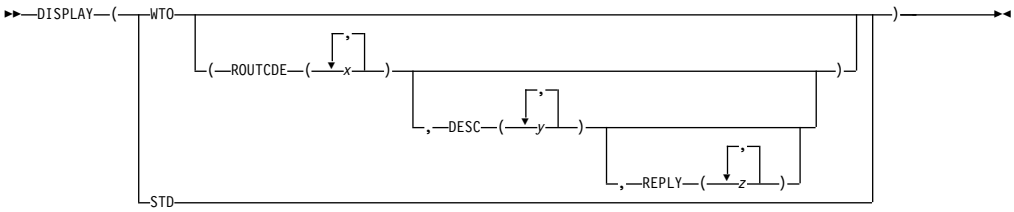
The purpose and usage of the DEPRECATENEXT option are the same as the DEPRECATE option, except that the compiler issues warning messages rather than error messages for items that you will deprecate in a future development phase.

Default: DEPRECATENEXT(BUILTIN() ENTRY() INCLUDE() STMT() VARIABLE())

For detailed information, see “DEPRECATE” on page 31.

DISPLAY

The DISPLAY option determines how the DISPLAY statement performs I/O.



STD
All DISPLAY statements are completed by writing the text to `stdout` and reading any REPLY text from `stdin`.

WTO
All DISPLAY statements without REPLY are completed by WTOs, and all DISPLAY statements with REPLY are completed by WTORs. This is the default.

The following suboptions are supported:

ROUTCDE
Specifies one or more values to be used as the ROUTCDE in the WTO. The default ROUTCDE is 2.

DESC
Specifies one or more values to be used as the DESC in the WTO. The default DESC is 3.

REPLY
Specifies one or more values to be used as the DESC in the WTOR. If omitted, the value from the DESC option (or the default) is used.

All values specified for the ROUTCDE, DESC and REPLY must between 1 and 16.

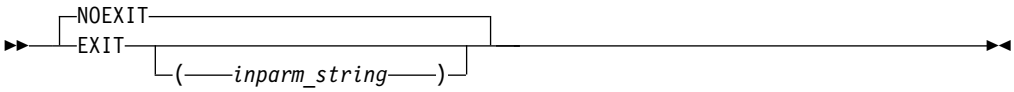
DLLINIT

The DLLINIT option applies OPTIONS(FETCHABLE) to all external procedures that are not MAIN. Use this option only on compilation units containing one external procedure, and then that procedure should be linked as a DLL.

NODLLINIT has no effect on your programs.

EXIT

The EXIT option enables the compiler user exit to be invoked.

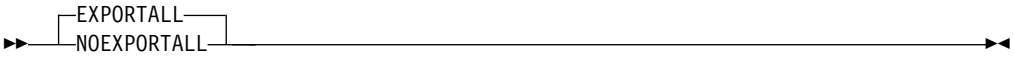


inparm_string
A string that is passed to the compiler user exit routine during initialization. The string can be up to 31 characters long.

See Chapter 24, "Using user exits," on page 483 for more information about how to use this option.

EXPORTALL

The EXPORTALL option controls whether to export all externally defined procedures and variable so that a DLL application can use them.



Under EXPORTALL, all externally defined procedures and variables are exported except those that specify OPTION(DLLINTERNAL).

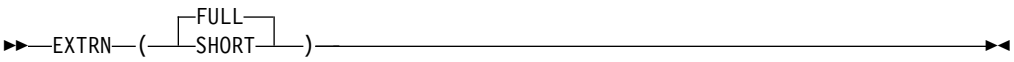
Under NOEXPORTALL, no externally defined procedures and variables are exported except those that specify OPTION(DLLEXTERNAL) or OPTIONS(FETCHABLE).

When linking a module as a DLL that will be simply fetched (rather than accessed through DLL functions), you must use the NOEXPORTALL option.

The default is EXPORTALL, but NOEXPORTALL is better for performance.

EXTRN

The EXTRN option controls when EXTRNs are emitted for external entry constants.

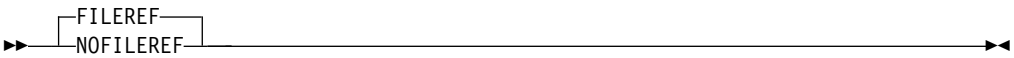


FULL
Emits EXTRNs for all declared external entry constants. This is the default.

SHORT
Emits EXTRNs only for those constants that are referenced.

FILEREF

The (NO)FILEREF option controls whether the compiler produces a file reference table. The NOFILEREF option eliminates the file reference table from the listing if the compiler issues no messages for the specified FLAG setting.



The default is FILEREF.

FLAG

The FLAG option specifies the minimum severity of error that requires a message listed in the compiler listing.

ABBREVIATION: F

I List all messages.

W List all except information messages.

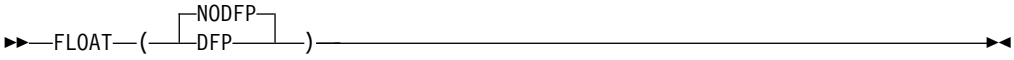
E List all except warning and information messages.

S List only severe error and unrecoverable error messages.

If messages are below the specified severity or are filtered out by a compiler exit routine, they are not listed.

FLOAT

The FLOAT option controls the use of the additional floating-point registers and whether Decimal Floating Point is supported.



DFP
The DFP facility is used: all DECIMAL FLOAT data will be held in the DFP format as described in the *z/Architecture Principles of Operation* manual, and operations using DECIMAL FLOAT will be carried using the DFP hardware instructions described therein.

NODFP
The DFP facility is not used.

Under FLOAT(DFP), the following applies:

- The maximum precision for extended DECIMAL FLOAT will be 34 (not 33 as it is for hex float).
- The maximum precision for short DECIMAL FLOAT will be 7 (not 6 as it is for hex float).
- The values for DECIMAL FLOAT for the following built-ins will all have the appropriate changes:
  - EPSILON
  - HUGE
  - MAXEXP
  - MINEXP
  - PLACES
  - RADIX
  - TINY
- The following built-in functions will all return the appropriate values for DECIMAL FLOAT (and values that will be much easier for a human to understand; for example, SUCC(1D0) will be 1.000_000_000_000_001, and the ROUND function will round at a decimal place of course.)
  - EXPONENT

  - PRED
  - ROUND
  - SCALE
  - SUCC
- Decimal floating-point literals, when converted to "right-units-view", that is, when the exponent has been adjusted, if needed, so that no nonzero digits follow the decimal point (for example, as would be done when viewing 3.1415E0 as 31415E-4), must have an exponent within the range of the normal numbers for the precision given by the literal. These bounds are given by the value of MINEXP-1 and MAXEXP-1. In particular, the following must hold:
  - For short float, -95 <= exponent <= 90
  - For long float, -383 <= exponent <= 369
  - For extended float, -6143 <= exponent <= 6111
- When a DECIMAL FLOAT is converted to CHARACTER, the string will hold 4 digits for the exponent (as opposed to the 2 digits used for hexadecimal float).
- The IEEE and HEXADEC attributes will be accepted only if applied to FLOAT BIN, and the DEFAULT(IEEE/HEXADEC) option will apply only to FLOAT BIN.
- The mathematical built-in functions (ACOS, COS, SQRT, etc) will accept DECIMAL FLOAT arguments and use corresponding Language Environment functions to evaluate them. DECIMAL FLOAT exponentiation will be handled in a similar way.
- Users of DFP need to be wary of the conversions that will arise in operations where one operand is FLOAT DECIMAL and the other is binary (that is, FIXED BINARY, FLOAT BINARY or BIT). In such operations, the PL/I language rules dictate that the FLOAT DECIMAL operand be converted to FLOAT BINARY, and that conversion will require a library call. For example, for an assignment of the form *A = A + B;*, where *A* is FLOAT DECIMAL and *B* is FIXED BINARY, three conversions will be done and two of those will be library calls:
  1. *A* will be converted through library call from FLOAT DECIMAL to FLOAT BINARY.
  2. *B* will be converted through inline code from FIXED BINARY to FLOAT BINARY.
  3. The sum *A + B* will be converted through library call from FLOAT BINARY to FLOAT DECIMAL.

  The use of the DECIMAL built-in function might help here: if the statement is changed to *A = A + DEC(B);*, the library calls will be eliminated. The library calls can also be eliminated by assigning *B* to a FLOAT DECIMAL temporary variable and then adding that temporary variable to *A*.
- The built-in function SQRTF is not supported for DECIMAL FLOAT arguments (because there is no hardware instruction to which it can be mapped).
- DFP is not supported by the CAST type function.

FLOATINMATH

The FLOATINMATH option specifies that the precision that the compiler should use when invoking the mathematical built-in functions.

ASIS
Arguments to the mathematical built-in functions will not be forced to have long or extended floating-point precision.

LONG
Any argument to a mathematical built-in function with short floating-point precision will be converted to the maximum long floating-point precision to yield a result with the same maximum long floating-point precision.

EXTENDED
Any argument to a mathematical built-in function with short or long floating-point precision will be converted to the maximum extended floating-point precision to yield a result with the same maximum extended floating-point precision.

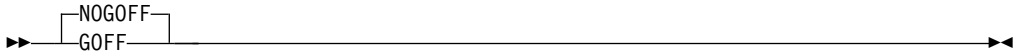
A FLOAT DEC expression with precision *p* has short floating-point precision if *p* <= 6, long floating-point precision if 6 < *p* <= 16, and the expression has extended floating-point precision if *p* > 16.

A FLOAT BIN expression with precision *p* has short floating-point precision if *p* <= 21, long floating-point precision if 21 < *p* <= 53, and the expression has extended floating-point precision if *p* > 53.

The maximum extended floating-point precision depends on the platform.

GOFF

The GOFF option instructs the compiler to produce an object file in the Generalized Object File Format (GOFF).



When the GOFF and OBJECT options are in effect, the compiler produces an object file in GOFF format.

When the NOGOFF and OBJECT options are in effect, the compiler produces an object file in XOBJ format.

The GOFF format supersedes the S/370 Object Module and Extended Object Module formats. It removes various limitations of the previous format (for example, 16 MB section size) and provides a number of useful extensions, including native z/OS support for long names and attributes. GOFF incorporates some aspects of industry standards such as XCOFF and ELF.

When you specify the GOFF option, you must use the binder to bind the output object.

The following options are not supported with the GOFF option:

- COMMON
- NOWRITABLE(PRV)

Note: When using GOFF and source files with duplicate file names, the linker might emit an error and discard one of the code sections. In this case, turn off the CSECT option by specifying NOCSECT.

GONUMBER

The GONUMBER option specifies that the compiler produces additional information that allows line numbers from the source program to be included in runtime messages.

ABBREVIATIONS: GN, NGN

SEPARATE
Places the generated statement number table in the separate debug file if the TEST(SEPARATE) option is specified. When you use GONUMBER(SEPARATE), the statement numbers are not available for inclusion in runtime messages.

NOSEPARATE
Places the generated statement number table in the object deck.

Alternatively, the line numbers can be derived by using the offset address, which is always included in runtime messages, and either the table produced by the OFFSET option or the assembler listing produced by the LIST option.

GONUMBER is forced by the ALL and STMT suboptions of the TEST option.

Note that the GOSTMT option does not exist. The only option that produces information at run time identifying where an error has occurred is the GONUMBER option. When the GONUMBER option is used, the term statement in the runtime error messages refers to the line numbers as used by the NUMBER compiler option, even if the STMT option is in effect.

If the GONUMBER(SEPARATE) option is specified without TEST(SEPARATE), it is changed to GONUMBER(NOSEPARATE).

If both TEST and NOGONUMBER are specified, the NOGONUMBER option is changed to GONUMBER(NOSEPARATE).

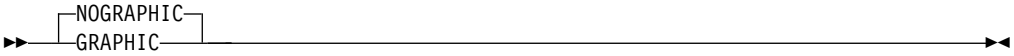
The default is NOGONUMBER.

For compatibility, when the GONUMBER option is specified, the default suboption is SEPARATE.

GRAPHIC

The GRAPHIC option specifies that the source program can contain double-byte characters.

The hexadecimal code '0E' is treated as the shift-out control code, and '0F' is treated as the shift-in control code, wherever they appear in the source program, including occurrences in comments and string constants.



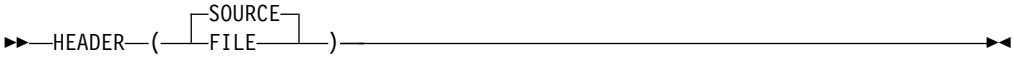
ABBREVIATIONS: GR, NGR

The GRAPHIC option will also cause the GRAPHIC ENVIRONMENT option to be applied to any STREAM file used in the compilation.

The GRAPHIC option must be specified if the source program uses any of the following:
- DBCS identifiers
- Graphic string constants
- Mixed-string constants
- Shift codes anywhere else in the source

HEADER

The HEADER option lets you control what appears in the middle of each header line in the compiler listing.



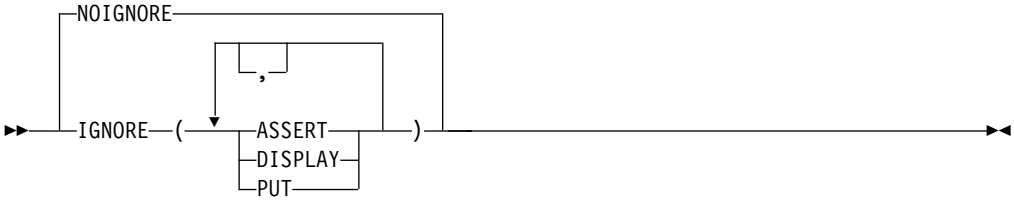
Specifying HEADER(SOURCE) causes the compiler to use the text in the first source line after any *PROCESS statements as the middle of each header line in the compiler listing.

Specifying HEADER(FILE) causes the compiler to use the name of the source file as the middle of each header line in the compiler listing.

The default is HEADER(SOURCE).

IGNORE

The IGNORE option controls whether ASSERT, DISPLAY, and PUT statements are ignored. When a statement is ignored, it is as if the statement is replaced by a semicolon.



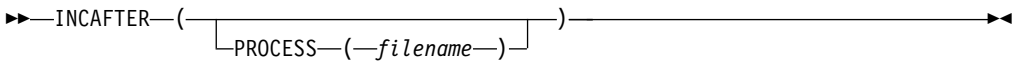
ASSERT
The compiler ignores all ASSERT statements, including any function references contained in those statements.

DISPLAY
The compiler ignores all DISPLAY statements, including any function references contained in those statements.

PUT
The compiler ignores all PUT FILE statements.

INCAFTER

The INCAFTER option specifies a file to be included after a particular statement in your source program.



filename
Name of the file to be included after the last PROCESS statement

Currently, PROCESS is the only suboption and specifies the name of a file to be included after the last PROCESS statement.

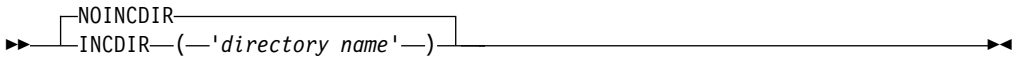
Consider the following example:

```
INCAFTER(PROCESS(DFTS))
```

This example is equivalent to having the statement `%INCLUDE DFTS;` after the last PROCESS statement in your source.

INCDIR

The INCDIR compiler option specifies a directory to be added to the search path used to locate include files.



directory name
Name of the directory that should be searched for include files. You can specify the INCDIR option more than once and the directories are searched in order.

Except under batch, the compiler looks for INCLUDE files in the following order:
1. Current® directory
2. Directories specified with the –I flag or with the INCDIR compiler option
3. /usr/include directory
4. PDS specified with the INCPDS compiler option

Under batch, this option is probably best used with the DFT(LOWERINC) option, and it affects only include statements of the form `%include x;`. For these include statements, an hfs file with the name `x.inc` will be sought first in the directories specified in this option. If the hfs file is not found, `x` must be a member of the pds(e) specified in the syslib dd. For include statements of the form `%include dd(x);`, no hfs file will ever be included: the member `x` must always be a member of the pds named by the specified dd.

INCLUDE

The INCLUDE compiler option controls whether the final pass of the compiler handles %INCLUDE and %XINCLUDE statements.

INCLUDE
Both the MACRO preprocessor and the final pass of the compiler handle %INCLUDE and %XINCLUDE statements.

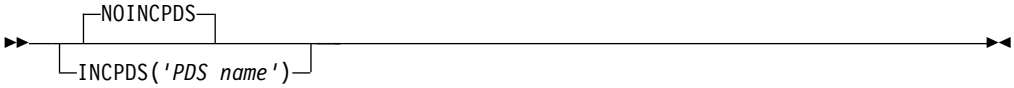
NOINCLUDE
Only the MACRO preprocessor handles %INCLUDE and %XINCLUDE statements.

INCLUDE is the default.

INCPDS

The INCPDS option specifies a PDS from which the compiler will include files when compiling a program under z/OS UNIX.

Note: This option applies only to compilations under z/OS UNIX.



PDS name
Name of a PDS from which files will be included

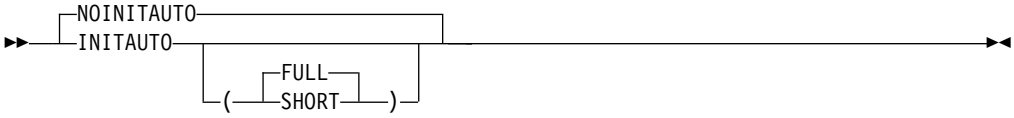
For example, if you want to compile the program TEST from a PDS named SOURCE.PLI and want to use the INCLUDE files from the PDS SOURCE.INC, you can specify the following command:

```
pli -c -qincpds="SOURCE.INC" "//'SOURCE.PLI(TEST)'"
```

The compiler looks for INCLUDE files in the following order:
1. Current directory
2. Directories specified with the –I flag or with the INCDIR compiler option
3. `/usr/include` directory
4. PDS specified with the INCPDS compiler option

INITAUTO

The INITAUTO option directs the compiler to add an INITIAL attribute to any AUTOMATIC variable declared without an INITIAL attribute.



Under INITAUTO(FULL), the compiler adds an INITIAL attribute to any AUTOMATIC variable that does not have an INITIAL attribute according to its data attributes:

- INIT((*) 0) if it is FIXED or FLOAT
- INIT((*) '') if it is PICTURE, CHAR, BIT, GRAPHIC or WIDECHAR
- INIT((*) SYSNULL()) if it is POINTER or OFFSET
- INIT((*) NULLENTRY()) if it is ENTRY

The compiler will not add an INITIAL attribute to variables with other attributes.

INITAUTO will cause more code to be generated in the prologue for each block containing any AUTOMATIC variables that are not fully initialized (but unlike the DFT(INITFILL) option, those variables will now have meaningful initial values) and will have a negative impact on performance.

The INITAUTO option does not apply an INITIAL attribute to any variable declared with the NOINIT attribute.

Under INITAUTO(SHORT), the compiler adds an INITIAL attribute to an AUTOMATIC variable that does not have an INITIAL attribute only if it is also a scalar and has one of the following attributes:

- POINTER
- OFFSET
- FIXED BIN
- FLOAT
- NONVARYING BIT
- NONVARYING CHAR(1)
- NONVARYING WCHAR(1)

As under INITAUTO(FULL), the added INITIAL attribute will be appropriate to the data type.

If you are using the runtime STORAGE option to zero-out all storage, the optimizer might still generate undesired code for uninitialized AUTOMATIC variables and especially for those that are optimized to registers. Scalar variables with the data types listed above are those most likely to be optimized to registers, and hence using INITAUTO(SHORT) might have less of a performance impact than using DFT(INITFILL); however, the latter would leave the optimizer with no ambiguous code. Moreover, your code is correct only if all variables are explicitly initialized before use.

INITBASED

The INITBASED option directs the compiler to add an INITIAL attribute to any BASED variable declared without an INITIAL attribute.

This option performs the same function as INITAUTO except for BASED variables.

The INITBASED option will cause more code to be generated for any ALLOCATE of a BASED variable that is not fully initialized and will have a negative impact on performance.

The INITBASED option does not apply an INITIAL attribute to any variable declared with the NOINIT attribute.

INITCTL

The INITCTL option directs the compiler to add an INITIAL attribute to any CONTROLLED variable declared without an INITIAL attribute.

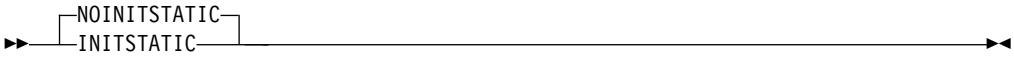
This option performs the same function as INITAUTO except for CONTROLLED variables.

The INITCTL option will cause more code to be generated for any ALLOCATE of a CONTROLLED variable that is not fully initialized and will have a negative impact on performance.

The INITCTL option does not apply an INITIAL attribute to any variable declared with the NOINIT attribute.

INITSTATIC

The INITSTATIC option directs the compiler to add an INITIAL attribute to any STATIC variable declared without an INITIAL attribute.



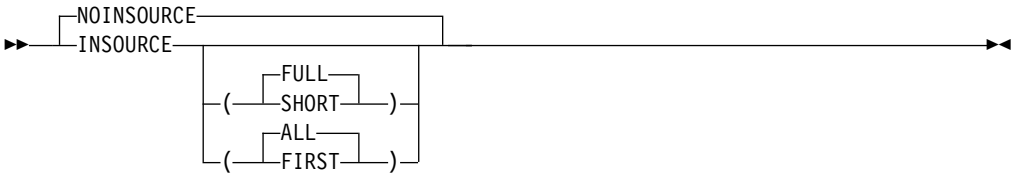
This option performs the same function as INITAUTO except for STATIC variables.

The INITSTATIC option could make some objects larger and some compilations longer, but should otherwise have no impact on performance.

The INITSTATIC option does not apply an INITIAL attribute to any variable declared with the NOINIT attribute.

INSOURCE

The INSOURCE option specifies that the compiler should include a listing of the source program before the PL/I macro, CICS, or SQL preprocessors translate it.



ABBREVIATION: IS, NIS

FULL
The INSOURCE listing will ignore %NOPRINT statements and will contain all the source before the preprocessor translates it.

FULL is the default.

SHORT
The INSOURCE listing will heed %PRINT and %NOPRINT statements.

ALL
The INSOURCE listing includes the source listings that are generated by each of the preprocessors and by the compiler itself. ALL is the default.

FIRST
The INSOURCE listing includes only the source listing that is generated by the first preprocessor.

Under the INSOURCE option, text is included in the listing not according to the logic of the program, but as each file is read. For example, consider the following simple program, which has a %INCLUDE statement between the PROC and END statements.

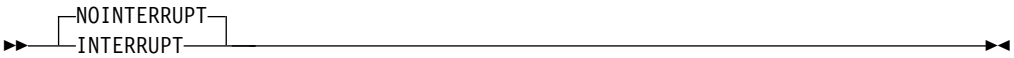
```
insource: proc options(main);
  %include member;
end;
```

The INSOURCE listing will contain all of the main program before any of the included text from the file `member` (and it will contain all of that file before any text included by it - and so on).

Under the INSOURCE(SHORT) option, text included by a %INCLUDE statement inherits the print/noprint status that was in effect when the %INCLUDE statement was executed, but that print/noprint status is restored at the end of the included text (however, in the SOURCE listing, the print/noprint status is not restored at the end of the included text).

INTERRUPT

The INTERRUPT option causes the compiled program to respond to attention requests (interrupts).



ABBREVIATION: INT, NINT

This option determines the effect of attention interrupts when the compiled PL/I program runs under an interactive system. This option will have an effect only on programs running under TSO. If you have written a program that relies on raising the ATTENTION condition, you must compile it with the INTERRUPT option. This option allows attention interrupts to become an integral part of programming. This gives you considerable interactive control of the program.

If you specify the INTERRUPT option, an established ATTENTION ON-unit gets control when an attention interrupt occurs. When the execution of an ATTENTION ON-unit is complete, control returns to the point of interrupt unless directed elsewhere by a GOTO statement. If you do not establish an ATTENTION ON-unit, the attention interrupt is ignored.

If you specify NOINTERRUPT, an attention interrupt during a program run does not give control to any ATTENTION ON-units.

If you require the attention interrupt capability only for testing purposes, use the TEST option instead of the INTERRUPT option.

Related information:

"TEST" on page 88
The TEST option specifies the level of testing capability that the compiler generates as part of the object code. You can use this option to control the location of test hooks and to control whether to generate a symbol table.

Chapter 22, "Interrupts and attention processing," on page 477

JSON

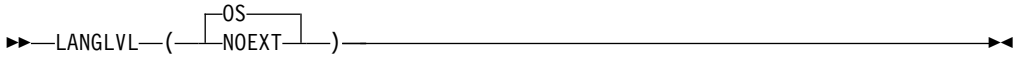
Using the JSON option, you can choose the case of the names in the JSON text generated by the JSONPUT built-in functions and expected by the JSONGET built-in functions.

CASE(UPPER | ASIS)
Under the CASE(UPPER) suboption, the names in the JSON text generated by the JSONPUT built-in functions and expected by the JSONGET built-in functions will all be in upper case.

Under the CASE(ASIS) suboption, the names in the JSON text generated by the JSONPUT built-in functions and expected by the JSONGET built-in functions will be in the case used in their declares. Note that if you use the MACRO preprocessor without using the macro preprocessor option CASE(ASIS), the source seen by the compiler will have all the names in upper case, which makes specifying the JSON(CASE(ASIS)) option useless.

LANGLVL

The LANGLVL option specifies the level of PL/I language definition that you want the compiler to accept.



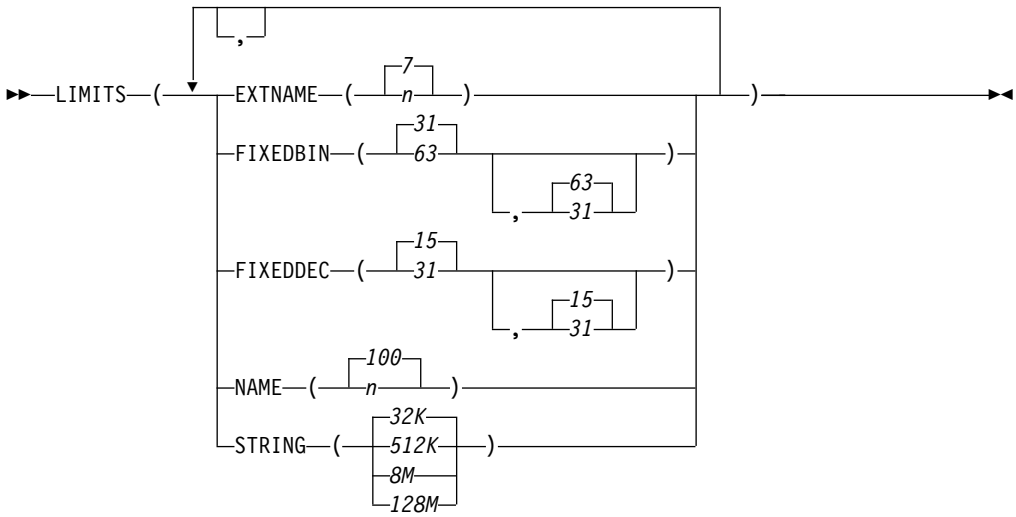
NOEXT
Only the following ENVIRONMENT options are accepted:

Bkwd	Genkey	Keyloc	Relative
Consecutive	Graphic	Organization	Scalarvarying
Ctlasa	Indexed	Recsize	Vsam
Deblock	Keylength	Regional	

OS All ENVIRONMENT options are allowed. For a complete list of the ENVIRONMENT options, see Table 15 on page 228.

LIMITS

The LIMITS option specifies various implementation limits.



EXTNAME
Specifies the maximum length for the EXTERNAL name. The maximum value for *n* is 100; the minimum value is 7.

FIXEDDEC
Specifies the maximum precision for FIXED DECIMAL to be either 15 or 31. The default is FIXEDDEC(15,31).

If FIXEDDEC(15,31) is specified, you can declare FIXED DECIMAL variables with precision greater than 15, but unless an expression contains an operand with precision greater than 15, the compiler uses 15 as the maximum precision for all arithmetic.

FIXEDDEC(15,31) will provide much better performance than FIXEDDEC(31).

FIXEDDEC(15) and FIXEDDEC(15,15) are equivalent; similarly, FIXEDDEC(31) and FIXEDDEC(31,31) are equivalent.

FIXEDDEC(31,15) is not allowed.

FIXEDBIN
Specifies the maximum precision for SIGNED FIXED BINARY to be either 31 or 63. The default is (31,63).

If FIXEDBIN(31,63) is specified, you can declare 8-byte integers, but unless an expression contains an 8-byte integer, the compiler uses 4-byte integers for all integer arithmetic.

Note, however, that specifying the FIXEDBIN(31,63) or FIXEDBIN(63) option might cause the compiler to use 8-byte integer arithmetic for expressions mixing data types. For example, if a FIXED BIN(31) value is added to a FIXED DEC(13) value, the compiler will produce a FIXED BIN result, and under LIMITS(FIXEDBIN(31,63)) that result would have a precision greater than 31 (because the FIXED DEC precision is greater than 9). When this occurs, the compiler will issue informational message IBM2809.

FIXEDBIN(31,63) will provide much better performance than FIXEDBIN(63).

FIXEDBIN(31) and FIXEDBIN(31,63) are equivalent; similarly, FIXEDBIN(63) and FIXEDBIN(63,63) are equivalent.

FIXEDBIN(63,31) and FIXEDBIN(31,31) are not allowed.

The maximum precision for UNSIGNED FIXED BINARY is one greater, that is, 32 and 64.

NAME
Specifies the maximum length of variable names in your program. The maximum value for *n* is 100; the minimum value is 31.

STRING
Accepts these values as the threshold for the length of a BIT, CHARACTER, or WIDECHAR variable: 32K, 512K, 8M, and 128M. This means that the length must be less than the threshold, and so the corresponding limits are 32767, 524287, 8388607, and 134217727.

- 32K is the default value. A larger value is accepted only if the CMPAT(V3) and BIFPREC(31) options are also specified.
- This limit applies to NONVARYING, VARYINGZ, and VARYING4, but not to VARYING. The maximum value allowed for VARYING is 32K.

LINECOUNT

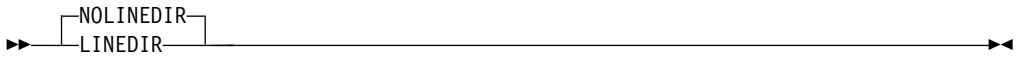
The LINECOUNT option specifies the number of lines per page for compiler listings, including blank and heading lines.

ABBREVIATION: LC

n The number of lines in a page in the listing. The value range is 10 - 32767.

LINEDIR

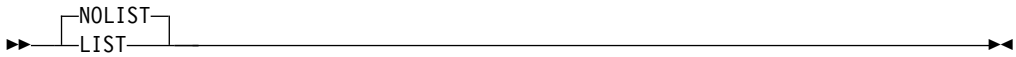
The LINEDIR option specifies that the compiler should accept %LINE directives.



If the LINEDIR option is specified, the compiler will reject all %INCLUDE statements. If the LINEDIR option is specified, the compiler will also reject the use of the SEPARATE suboption of the TEST option.

LIST

The LIST option specifies that the compiler should produce a pseudo-assembler listing.



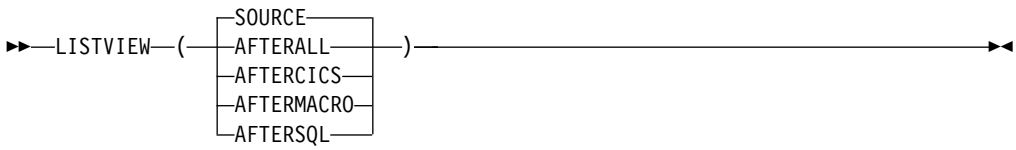
Specifying the LIST option will increase the time and region required for a compilation. The OFFSET and MAP options can provide the information you need at much less cost.

The pseudo-assembler listing will also include at the end of the listing for each block the offset of the first instruction in that block from the start of the whole compilation unit.

LISTVIEW

The LISTVIEW option specifies whether the compiler should show the source in the source listing or whether it should show the source after it has been processed by one or more of the preprocessors.

The LISTVIEW option is ignored if the NOSOURCE option is in effect.



SOURCE
Causes the source listing to show the unadulterated source and, more importantly perhaps, it will cause IBM Debug Tool to bring up this as the source view.

AFTERALL
Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the last preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view if the SEPARATE suboption of the TEST compiler option is also specified.

AALL can be used as an abbreviation for AFTERALL.

AFTERCICS
Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the CICS preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view if the SEPARATE suboption of the TEST compiler option is also specified.

ACICS can be used as an abbreviation for AFTERCICS.

AFTERMACRO
Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the MACRO preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view if the SEPARATE suboption of the TEST compiler option is also specified.

AMACRO can be used as an abbreviation for AFTERMACRO.

AFTERSQL
Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the SQL preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view if the SEPARATE suboption of the TEST compiler option is also specified.

ASQL can be used as an abbreviation for AFTERSQL.

If the TEST option is specified and a suboption other than SOURCE is specified for LISTVIEW, the SEPARATE suboption must also be specified for the TEST option.

The following example shows the differing effects of the AFTERMACRO, AFTERSQL, and AFTERALL suboptions.

Suppose the PP option is PP(MACRO('INCONLY'), SQL, MACRO).

- Under LISTVIEW(AFTERMACRO), the "source" in the listing and in the Debug Tool source window if TEST(SEP) were specified would appear as if it came from the MDECK that the second invocation of the MACRO preprocessor would have produced.
- Under LISTVIEW(AFTERSQL), the "source" in the listing and in the Debug Tool source window if TEST(SEP) were specified would appear as if it came from the MDECK that the invocation of the SQL preprocessor would have produced (and hence %DCL and other macro statements would still be visible).
- Under LISTVIEW(AFTERALL), the "source" would be as under the LISTVIEW(AFTERMACRO) option because the MACRO preprocessor is the last in the PP option.

LP

The LP option specifies whether the compiler generates 31-bit code or 64-bit code. It also determines the default size of POINTER and HANDLE and related variables.

32 Under LP(32), the compiler generates 31-bit code. In addition, type *size_t* resolves to FIXED BIN(31). The default size of POINTER and HANDLE is four bytes.

64 Under LP(64), the compiler generates 64-bit code. In addition, type *size_t* resolves to FIXED BIN(63). The default size of POINTER and HANDLE is eight bytes.

Note: Under LP(64), some compiler options are not applicable. For more information, see "Using compiler options to build 64-bit applications" on page 195.

The default is LP(32).

Related information:

Chapter 7, "Considerations for developing 64-bit applications," on page 195
You can use Enterprise PL/I to develop 31-bit or 64-bit applications. For your applications to support the 64-bit environment, you might need to adapt your code as appropriate. This section describes considerations in development and compilation that you must take into account.

Chapter 6, "Link-editing and running for 64-bit programs," on page 187
After compilation with LP(64), your 64-bit program consists of one or more object modules that contain unresolved references to each other, as well as references to the Language Environment runtime library. These references are resolved during link-editing (statically) or during execution (dynamically).

MACRO

The MACRO option invokes the MACRO preprocessor.

ABBREVIATIONS: M, NM

You can also invoke the MACRO preprocessor through the PP(MACRO) option. However, the use of both the MACRO option and the PP(MACRO) option in the same compilation is not recommended.

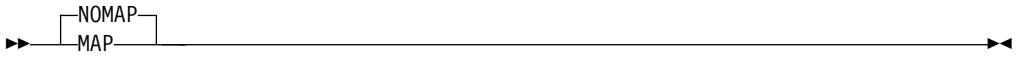
Related information:

"PP" on page 61
The PP option specifies which (and in what order) preprocessors are invoked before compilation.

"Macro preprocessor" on page 118
Macros allow you to write commonly used PL/I code in a way that hides implementation details and the data that is manipulated and exposes only the operations. In contrast with a generalized subroutine, macros allow generation of only the code that is needed for each individual use. You can invoke the macro preprocessor by specifying either the MACRO option or the PP(MACRO) option.

MAP

The MAP option specifies that the compiler produces additional information that can be used to locate static and automatic variables in dumps.



MARGINI

The MARGINI option specifies a character that the compiler will place in the column preceding the left-hand margin, and also in the column following the right-hand margin, of the listings produced by the INSOURCE and SOURCE options.

ABBREVIATIONS: MI, NMI

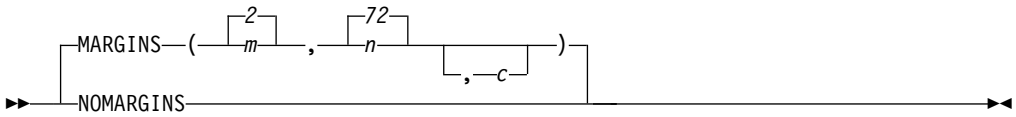
c The character to be printed as the margin indicator

Note: NOMARGINI is equivalent to MARGINI(' ').

MARGINS

The MARGINS option specifies which part of each compiler input record contains PL/I statements, and the position of the ANS control character that formats the listing, if the SOURCE option, the INSOURCE option, or both apply. The compiler does not process data that is outside these limits, but it does include it in the source listings.

The PL/I source is extracted from the source input records so that the first data byte of a record immediately follows the last data byte of the previous record. For variable records, you must ensure that when you need a blank, you explicitly insert it between margins of the records.



ABBREVIATION: MAR

m The column number of the leftmost character (first data byte) that is processed by the compiler. It must not exceed 100.

n The column number of the rightmost character (last data byte) that is

processed by the compiler. It should be greater than *m*, but must not exceed 200, except under MVS batch where it must not exceed 100.

Variable-length records are effectively padded with blanks to give them the maximum record length.

c The column number of the ANS printer control character. It must not exceed 200, except under MVS batch where it must not exceed 100, and it should be outside the values specified for *m* and *n*. A value of 0 for *c* indicates that no ANS control character is present. Only the following control characters can be used:

(blank)
Skip one line before printing

0 Skip two lines before printing

– Skip three lines before printing

+ No skip before printing

1 Start new page

Any other character is an error and is replaced by a blank.

Do not use a value of *c* that is greater than the maximum length of a source record, because this causes the format of the listing to be unpredictable. To avoid this problem, put the carriage control characters to the left of the source margins for variable-length records.

Specifying MARGINS(,,c) is an alternative to using %PAGE and %SKIP statements (described in the *PL/I Language Reference*).

The IBM-supplied default for fixed-length records is MARGINS(2,72). For variable-length and undefined-length records, the IBM-supplied default is MARGINS(10,100). This specifies that there is **no** printer control character.

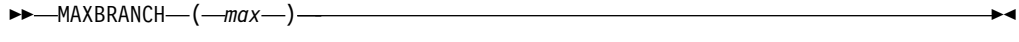
Use the MARGINS option to override the default for the primary input in a program. The secondary input must have the same margins as the primary input.

The NOMARGINS option will suppress any previously encountered instance of the MARGINS option. The purpose of this option is to allow your installation to have a default set of compile-time options that use a MARGINS option tailored for their fixed format source preferences while retaining the ability to use variable source format files.

You would usually specify the NOMARGINS option, if you use it at all, as part of the parameter-string passed to the compiler. The compiler will ignore NOMARGINS if it finds the option in a %PROCESS statement.

MAXBRANCH

The MAXBRANCH option flags blocks that have too many branches. Branches include all conditional jumps and each WHEN in a SELECT statement that can be turned into a branch table.



max
The limit that measures the cyclomatic or conditional complexity of the block. The default is 2000.

A statement of the form "`if a then ...; else ...`" adds 1 to the total number of branches in its containing block, and a statement of the form "`if a = 0 | b = 0 then ...`" adds 2.

MAXGEN

The MAXGEN option specifies the maximum number of intermediate language statements that should be generated for any user statement. The option will cause the compiler to flag any statement where this maximum is exceeded.

The number of intermediate language statements generated for any user statement might vary depending on the compiler release, the compiler maintenance level, and the compiler options in effect. This option is intended only to help you find statements for which excessive amounts of code are generated, which might indicate that they are perhaps are poorly coded.

However, note that using a preprocessor might cause the number of intermediate language statements generated for some statements to be very large. In such a situation, it might be better either to set the MAXGEN threshold to be larger or to use the LISTVIEW(AFTERALL) option.

The default is MAXGEN(100000).

MAXMEM

When you compile with OPTIMIZE, the MAXMEM option limits the amount of memory used for local tables of specific, memory-intensive optimizations to the specified number of kilobytes.

The range of memory that you can specify for MAXMEM is 1 - 2097152. The default is 1048576.

If you specify the maximum value of 2097152, the compiler will assume that unlimited memory is available. If you specify any smaller value for MAXMEM, the compiler, especially when the OPT(2) option is in effect, might issue a message saying that optimization is inhibited and that you should try using a larger value for MAXMEM.

Use the MAXMEM option if you know that less (or more) memory is available than implied by the default value.

If the memory specified by the MAXMEM option is insufficient for a particular optimization, the compilation is completed in such a way that the quality of the optimization is reduced, and a warning message is issued.

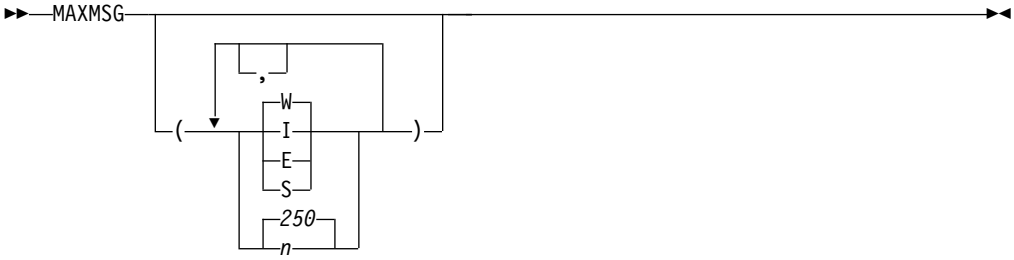
ABBREVIATIONS: MAXM

When a large size is specified for MAXMEM, compilation might be aborted because of insufficient virtual storage, depending on the source file being compiled, the size of the subprogram in the source, and the virtual storage available for the compilation.

The advantage of using the MAXMEM option is that, for large and complex applications, the compiler produces a slightly less-optimized object module and generates a warning message, instead of terminating the compilation with an error message of `insufficient virtual storage`.

MAXMSG

The MAXMSG option specifies the maximum number of messages with a given severity (or higher) that the compilation should produce.



I Count all messages.

W Count all except information messages.

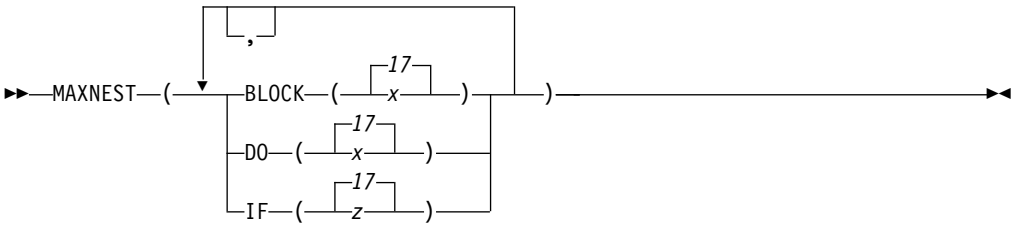
E Count all except warning and information messages.

S Count only severe error and unrecoverable error messages.

n Terminate the compilation if the number of messages exceeds this value. If messages are below the specified severity or are filtered out by a compiler exit routine, they are not counted in the number. Enter a value in the range 0 - 32767. If you specify 0, the compilation terminates when the first error of the specified severity is encountered.

MAXNEST

The MAXNEST option specifies the maximum nesting of various kinds of statements that are allowed before the compiler flags your program as too complex.



BLOCK
Specifies the maximum nesting of BEGIN and PROCEDURE statements.

DO Specifies the maximum nesting of DO statements.

IF Specifies the maximum nesting of IF statements.

The value range of any nesting limit is 1 - 50.

The default is MAXNEST(BLOCK(17) DO(17) IF(17)).

MAXSTMT

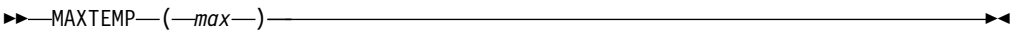
Under the MAXSTMT option, optimization will be turned off for any block that has more than the specified number of statements. Use the MAXSTMT option - with a reasonable limit to the number of statements - if you want the compiler to optimize the code generated for a program and are willing for the compiler to optimize only the reasonably sized blocks in that program.

When a large size is specified for MAXSTMT, if some blocks have a large number of statements, compilation might be aborted if there is not enough virtual storage available.

The default for MAXSTMT is 4096.

MAXTEMP

The MAXTEMP option determines when the compiler flags statements that are using an excessive amount of storage for compiler-generated temporaries.



max
The limit for the number of bytes that can be used for compiler-generated temporaries. The compiler flags any statement that uses more bytes than the amount specified by *max*. The default for *max* is 50000.

You should examine statements that are flagged under this option. If you code them differently, you might be able to reduce the amount of stack storage required by your code.

MDECK

The MDECK option specifies that the preprocessor produces a copy of its output either on the file defined by the SYSPUNCH DD statement under z/OS or on the .dek file under z/OS UNIX.

ABBREVIATIONS: MD, NMD

The MDECK option allows you to retain the output from the preprocessor as a file of 80-column records. This option is applicable only when the MACRO option is in effect.

AFTERALL
Causes the file to be generated after the invocation of the last preprocessor.

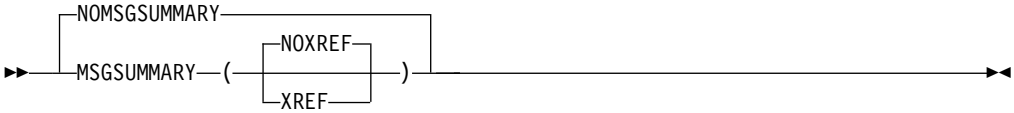
AFTERMACRO
Causes the file to be generated after the last invocation (if any) of the macro preprocessor.

Related information:

"MACRO" on page 49
The MACRO option invokes the MACRO preprocessor.

MSGSUMMARY

The MSGSUMMARY option determines whether the compiler adds a summary of all messages that are issued during compilation into the listing.



MSGSUMMARY(NOXREF)
The compiler adds a message summary to the listing. The summary is after the file reference table in the listing. It is sorted by compiler component and within each component by severity and then by message number.

The summary includes the following information:

- One instance of each message that is produced in the compilation
- The number of times that each message is produced

MSGSUMMARY(XREF)
The compiler adds a message summary to the listing. The summary is the same as the one added when MSGSUMMARY(NOXREF) is specified with one difference: after each message the summary lists all the line or statement numbers where the message is issued.

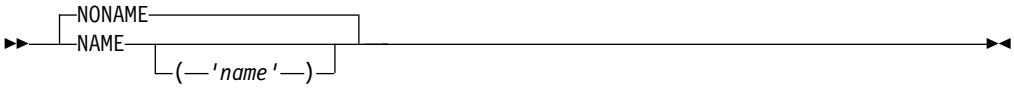
NOMSGSUMMARY
No message summary is produced.

NOMSGSUMMARY is the default. When MSGSUMMARY is specified, MSGSUMMARY(NOXREF) is the default.

For the compiler listing example with a message summary generated by using MSGSUMMARY, see Figure 4 on page 112.

NAME

The NAME option specifies that the TEXT file created by the compiler will contain a NAME record.



ABBREVIATIONS: N

If no name is specified as a suboption of the NAME option, then the name used is determined as follows:

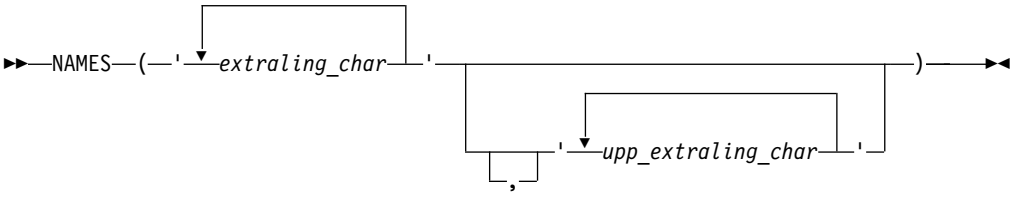
- If there is a PACKAGE statement, the leftmost name on it is used.
- Otherwise, the leftmost name on the first PROCEDURE statement is used.

The length of the name must not be greater than 8 characters if the LIMITS(EXTNAME(*n*)) option is used with *n* <= 8.

NAMES

The NAMES option specifies the *extralingual characters* that are allowed in identifiers.

Extralingual characters are those characters other than the 26 alphabetic, 10 digit, and special characters defined in the *PL/I Language Reference*.



extralingual_char
An extralingual character

upp_extraling_char
The extralingual character that you want interpreted as the uppercase version of the corresponding character in the first suboption

If you omit the second suboption, PL/I uses the character specified in the first suboption as both the lowercase and the uppercase values. If you specify the second suboption, you must specify the same number of characters as you specify in the first suboption.

The default is NAMES('#@$' '#@$').

NATLANG

The NATLANG option specifies the language for compiler messages, headers, and so on.

ENU
All compiler messages, headers, and so on will be in mixedcase English.

UEN
All compiler messages, headers, and so on will be in uppercase English.

NEST

The NEST option specifies that the listing resulting from the SOURCE option indicates the block level and the do-group level for each statement.

NOT

The NOT option specifies up to seven alternate symbols that can be used as the logical NOT operator.

char
A single SBCS character

You cannot specify any of the alphabetic characters, digits, and special characters defined in the *PL/I Language Reference*, except for the standard logical NOT symbol (¬). You must specify at least one valid character.

When you specify the NOT option, the standard NOT symbol is no longer recognized unless you specify it as one of the characters in the character string.

For example, NOT('~') means that the tilde character, 'A1'X, will be recognized as the logical NOT operator, and the standard NOT symbol, '¬', '5F'X, will not be recognized. Similarly, NOT('~¬') means that either the tilde or the standard NOT symbol will be recognized as the logical NOT operator.

The IBM-supplied default code point for the NOT symbol is '5F'X. The logical NOT sign might appear as a logical NOT symbol (¬) or a caret symbol (^) on your keyboard.

NULLDATE

The NULLDATE option instructs the compiler to accept the SQL null date as a valid date in some datetime handling built-in functions.

Under the NULLDATE option, the VALIDDATE and REPATTERN built-in functions will accept the SQL null date (with *year*, *day*, and *month* all equal to one) as a valid date.

The default is NONULLDATE.

NUMBER

The NUMBER option specifies that statements in the source program are to be identified by the line and file number of the file from which they derived, and that this pair of numbers is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, LIST, MAP, OFFSET, SOURCE, and XREF options.

The File Reference Table at the end of the listing shows the number assigned to each of the input files read during compilation.

If a preprocessor has been used, more than one line in the source listing might be identified by the same line and file numbers. For example, almost every EXEC CICS statement generates several lines of code in the source listing, but these are all identified by one line and file number.

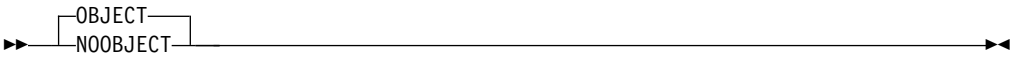
In the pseudo-assembler listing produced by the LIST option, the file number is left blank for the first file.

NUMBER and STMT are mutually exclusive and specifying one will negate the other.

The default is NUMBER.

OBJECT

The OBJECT option specifies that the compiler creates an object module. Under batch z/OS, the compiler stores the object in the data set defined by the SYSLIN DD, and underz/OS UNIX, the compiler creates a .o file.



ABBREVIATIONS: OBJ, NOBJ

Under the NOOBJECT option, the compiler does not create an object module. However, under the NOOBJECT option, the compiler will complete all of its syntactic and semantic analysis phases as well as all of its detection of uninitialized variables, and hence it could produce more messages than under the NOCOMPILE, NOSEMANTIC, or NOSYNTAX option.

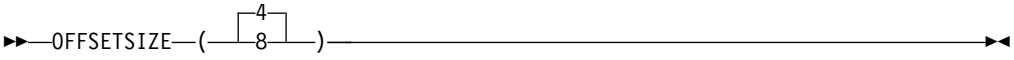
Under the NOOBJECT option, the LIST, MAP, OFFSET, and STORAGE options will be ignored.

OFFSET

The OFFSET option specifies that the compiler is to print a table of line numbers for each procedure and BEGIN block with their offset addresses relative to the primary entry point of the procedure. This table can be used to identify a statement from a runtime error message if the GONUMBER option is not used.

OFFSETSIZE

The OFFSETSIZE option determines the size of OFFSET variables in 64-bit applications.



4 Under OFFSETSIZE(4), all OFFSET variables are four bytes in size. This is the default.

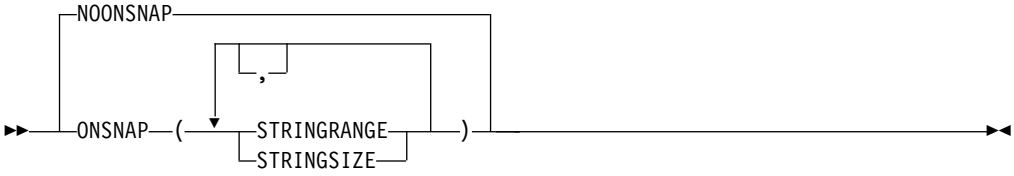
8 Under OFFSETSIZE(8), all OFFSET variables are eight bytes in size.

Any code that shares either OFFSET or AREA variables must be compiled with the same value for the OFFSETSIZE option.

The OFFSETSIZE option is ignored if the LP(32) option is in effect.

ONSNAP

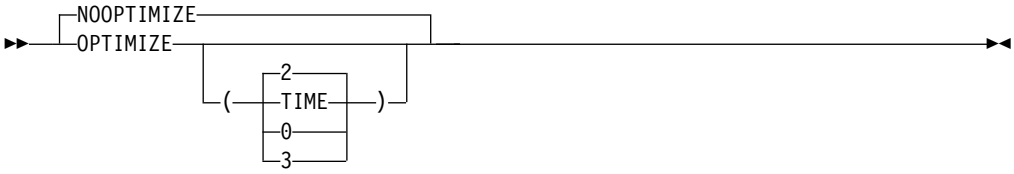
For a PROCEDURE with either the OPTIONS(MAIN) or the OPTIONS(FROMALIEN) attribute, the ONSNAP option specifies that the compiler should insert an `ON STRINGRANGE SNAP;` statement, an `ON STRINGSIZE SNAP;` statement, or both into the prologue code for that PROCEDURE. This can make it easier to determine the calling chain if the corresponding conditions are raised in other routines called from such a PROCEDURE.



The ONSNAP option has no affect on a PROCEDURE without either of these attributes.

OPTIMIZE

The OPTIMIZE option specifies the type of optimization required.



ABBREVIATIONS: OPT, NOPT

OPTIMIZE(0)
Specifies fast compilation speed, but inhibits optimization.

OPTIMIZE(2)
Optimizes the machine instructions generated to produce a more efficient object program. This type of optimization can also reduce the amount of main storage required for the object module.

OPTIMIZE(3)
Performs all the optimizations done under OPTIMIZE(2) plus some additional optimizations. Under OPTIMIZE(3), the compiler will generally, but especially for programs with large blocks and many variables, generate smaller and more efficient object code. However, it might also take considerably more time and region to complete compilations under OPTIMIZE(3) than under OPTIMIZE(2).

It is strongly recommended that the DFT(REORDER) option be used with the OPTIMIZE option. In fact, the effect of OPTIMIZE is severely limited for any PROCEDURE or BEGIN-block for which all of the following conditions are true:

- The ORDER option applies to the block.
- The block contains ON-units for hardware-detected conditions (such as ZERODIVIDE).
- The block has labels that are the (potential) target of branches out of those ON-units.

The use of OPTIMIZE(2) could result in a substantial increase in compile time over NOOPTIMIZE and a substantial increase in the space required. For example, compiling a large program at OPTIMIZE(2) might take several minutes and could require a region of 100M or more.

The use of OPTIMIZE(3) will increase the time and region needed for a compilation over what is needed under OPTIMIZE(2). For large programs, the time to compile a program under OPTIMIZE(3) can be more than twice the time needed under OPTIMIZE(2).

During optimization the compiler can move code to increase runtime efficiency. As a result, statement numbers in the program listing might not correspond to the statement numbers used in runtime messages.

NOOPTIMIZE is the equivalent of OPTIMIZE(0).

OPTIMIZE(TIME) is the equivalent of OPTIMIZE(2).

Note that the use of OPTIMIZE(2) or OPTIMIZE(3) severely limits the functionality of the TEST option, as follows:

- If the HOOK suboption of TEST is in effect, only block hooks will be generated.
- If the NOHOOK suboption of TEST is in effect, attempts to list or change a variable might fail (because the variable might have been optimized into a register), and attempts to stop at a particular statement might cause the debugger to stop several times (because the statement might have split up into several parts).

The use of the PREFIX option with one or more of the checkout conditions (SIZE, STRINGRANGE, STRINGSIZE, and SUBSCRIPTRANGE) can significantly increase the time and space needed for a compilation.

Related information:

Chapter 15, “Improving performance,” on page 329
Many considerations for improving the speed of your program are independent of the compiler that you use and the platform on which it runs. This chapter, however, identifies those considerations that are unique to the PL/I compiler and the code it generates.

OPTIONS

The OPTIONS option specifies that the compiler includes a list showing the compiler options to be used during this compilation in the compiler listing.

ABBREVIATIONS: OP, NOP

This list includes all options applied by default, those specified in the PARM parameter of an EXEC statement or in the invoking command (pli), those specified in a %PROCESS statement, those specified in the IBM_OPTIONS environment variable under z/OS, and all those incorporated from any options file.

Under OPTIONS(DOC), the OPTIONS listing will include only those options (and suboptions) documented in this document at the time of the compiler's release.

Under OPTIONS(ALL), the OPTIONS listing will also include any option added by PTF after the compiler's release.

OR

The OR option specifies up to seven alternate symbols as the logical OR operator. These symbols are also used as the concatenation operator, which is defined as two consecutive logical OR symbols.

Note: Do not code any blanks between the quotation marks.

The IBM-supplied default code point for the OR symbol (|) is '4F'X.

char
A single SBCS character

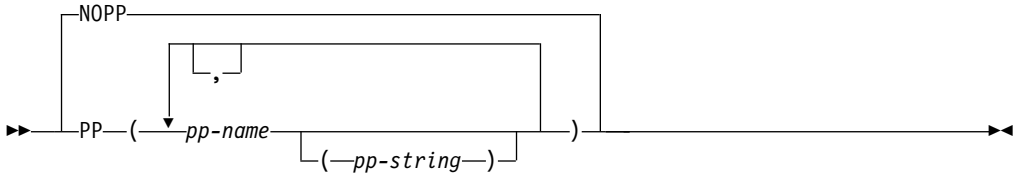
You cannot specify any of the alphabetic characters, digits, and special characters defined in the *PL/I Language Reference*, except for the standard logical OR symbol (|). You must specify at least one valid character.

If you specify the OR option, the standard OR symbol is no longer recognized unless you specify it as one of the characters in the character string.

For example, OR('\') means that the backslash character, 'E0'X, will be recognized as the logical OR operator, and two consecutive backslashes will be recognized as the concatenation operator. The standard OR symbol, '|', '4F'X, will not be recognized as either operator. Similarly, OR('\|') means that either the backslash or the standard OR symbol will be recognized as the logical OR operator, and either symbol or both symbols can be used to form the concatenation operator.

PP

The PP option specifies which (and in what order) preprocessors are invoked before compilation.



pp-name
The name given to a particular preprocessor. CICS, INCLUDE, MACRO and SQL are the only preprocessors currently supported. Using an undefined name causes a diagnostic error.

pp-string
A string, delimited by quotation marks, of up to 100 characters representing

the options for the corresponding preprocessor. For example, PP(MACRO('CASE(ASIS)')) invokes the MACRO preprocessor with the option CASE(ASIS).

Preprocessor options are processed from left to right, and if two options conflict, the last (rightmost) option is used. For example, if you invoke the MACRO preprocessor with the option string 'CASE(ASIS) CASE(UPPER)', then the option CASE(UPPER) is used.

You can specify a maximum of 31 preprocessor steps, and you can specify the same preprocessor more than once with the exception of the CICS and SQL preprocessors. The CICS preprocessor must be invoked at most once, and the SQL preprocessor must be invoked no more than twice. The SQL preprocessors can be invoked twice only if the first specification specifies INCONLY as its option.

If the MACRO option is specified along with the PP option, the MACRO preprocessor will be added to the beginning of the list of preprocessors in the PP option unless it is already the first in that list. So, specifying MACRO and PP(SQL MACRO) will cause the PP option to become PP(MACRO SQL MACRO), and the MACRO preprocessor will be invoked twice. However, specifying MACRO and PP(MACRO SQL) will leave the PP option unchanged, and the MACRO preprocessor will be invoked only once. However, the use of both the MACRO option and the PP(MACRO) option in the same compilation is not recommended.

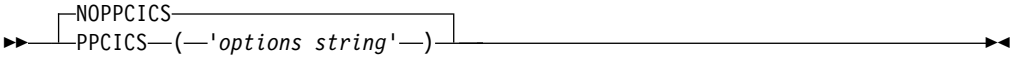
If you specify the PP option more than once, the compiler effectively concatenates them. So specifying PP(SQL) PP(CICS) is the same as specifying PP(SQL CICS). This also means that if you specifies PP(MACRO SQL('CCSID0')) and PP(MACRO SQL('CCSID0 DATE(ISO)')), the resulting PP option is PP(MACRO SQL('CCSID0') MACRO SQL('CCSID0 DATE(ISO)')), and both the MACRO preprocessor and the SQL preprocessor will be invoked twice, and the second invocation of the SQL preprocessor will be in error. If you do this to override the earlier SQL options, it might be better not to specify the preprocessor options in the PP option, but rather to specify them through the PPSQL option, that is, specify PP(MACRO SQL) PPSQL('CCSID0 DATE(ISO)').

Related information:

Chapter 2, “PL/I preprocessors,” on page 117
When you use the PL/I compiler, you can specify one or more of the integrated preprocessors in your program. You can specify the include preprocessor, the macro preprocessor, the SQL preprocessor, or the CICS preprocessor, and specify the order in which you want them to be called.

PPCICS

The PPCICS option specifies options to be passed to the CICS preprocessor if it is invoked.



Specifying PPCICS('EDF') PP(CICS) is the same as specifying PP(CICS('EDF')).

This option has no effect unless the PP(CICS) option is specified. However, if you want to specify a set of CICS preprocessor options that should be used if and when the CICS preprocessor is invoked, you can specify this option in the

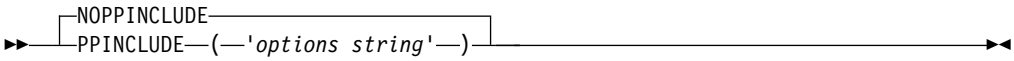
installation options exit. Then whenever you specify PP(CICS), the set of options specified in the PPCICS option will be used.

Also, any options specified when the preprocessor is invoked overrule those specified in the PPCICS option. So specifying PPCICS('EDF') PP(CICS('NOEDF')) is the same as specifying PP(CICS('EDF NOEDF')) or the even simpler PP(CICS('NOEDF')).

The options string is limited to 1000 characters in length. However, if the string is longer than 100 characters, it will not be shown in the options listing.

PPINCLUDE

The PPINCLUDE option specifies options to be passed to the INCLUDE preprocessor if it is invoked.



Specifying PPINCLUDE('ID(-inc)') PP(INCLUDE) is the same as specifying PP(INCLUDE('ID(-inc)')).

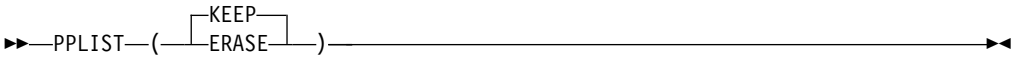
This option has no effect unless the PP(INCLUDE) option is specified. However, if you want to specify a set of INCLUDE preprocessor options that should be used if and when the INCLUDE preprocessor is invoked, you can specify this option in the installation options exit. Then whenever you specify PP(INCLUDE), the set of options specified in the PPINCLUDE option will be used.

Also, any options specified when the preprocessor is invoked overrule those specified in the PPINCLUDE option. So specifying PPINCLUDE('ID(-inc)') PP(INCLUDE('ID(+include)')) is the same as specifying PP(INCLUDE('ID(-inc) ID(+include)')) or the even simpler PP(INCLUDE('ID(+include)')).

The options string is limited to 1000 characters in length. However, if the string is longer than 100 characters, it will not be shown in the options listing.

PPLIST

The PPLIST option controls whether the compiler keeps or erases the part of the listing that is generated by each preprocessor phase.



When you specify PPLIST(KEEP), the compiler keeps the part of the listing that is generated by each preprocessor phase.

When you specify PPLIST(ERASE), the compiler erases the part of the listing that is generated by any preprocessor phase that produces no messages.

The compiler does not count messages that are suppressed by the EXIT and FLAG options. Therefore, specifying both FLAG(W) and PPLIST(ERASE) causes the compiler to suppress all output from any preprocessor that produces no warning, error, or severe messages.

PPLIST(KEEP) is the default.

PPMACRO

The PPMACRO option specifies options to be passed to the MACRO preprocessor if it is invoked.

Specifying PPMACRO('CASE(ASIS)') PP(MACRO) is the same as specifying PP(MACRO('CASE(ASIS)')).

This option has no effect unless the PP(MACRO) option is specified. However, if you want to specify a set of MACRO preprocessor options that should be used if and when the MACRO preprocessor is invoked, you can specify this option in the installation options exit. Then whenever you specify the MACRO or PP(MACRO) options, the set of options specified in the PPMACRO option will be used.

Also, any options specified when the preprocessor is invoked overrule those specified in the PPMACRO option. So specifying PPMACRO('CASE(ASIS)') PP(MACRO('CASE(UPPER)')) is the same as specifying PP(MACRO('CASE(ASIS) CASE(UPPER)')) or the even simpler PP(MACRO('CASE(UPPER)')).

The options string is limited to 1000 characters in length. However, if the string is longer than 100 characters, it will not be shown in the options listing.

PPSQL

The PPSQL option specifies options to be passed to the SQL preprocessor.

Specifying PPSQL('APOSTSQL') PP(SQL) is the same as specifying PP(SQL('APOSTSQL')).

This option has no effect unless the PP(SQL) option is specified. However, if you want to specify a set of options that should be used if the SQL preprocessor is invoked, you can specify this option in the installation options exit. Then, whenever you specify PP(SQL), the set of options in the PPSQL option are used.

Also, any options that are specified when the preprocessor is invoked overrule those that are specified in the PPSQL option. Therefore, specifying PPSQL('APOSTSQL') PP(SQL('QUOTESQL')) is the same as specifying PP(SQL('APOSTSQL QUOTESQL')) or the even simpler PP(SQL('QUOTESQL')).

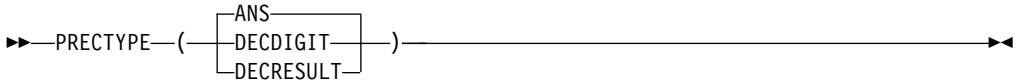
The *options string* is limited to 1000 characters in length. However, if the string is longer than 100 characters, it will not be shown in the options listing.

PPTRACE

The PPTRACE option specifies that when a deck file is written for a preprocessor, every nonblank line in that file is preceded by a line containing a %LINE directive. The directive indicates the original source file and line to which the nonblank line should be attributed.

PRECTYPE

The PRECTYPE option determines how the compiler derives the attributes for the MULTIPLY, DIVIDE, ADD and SUBTRACT built-in functions when the operands are FIXED and at least one is FIXED BIN.



ANS

Under PRECTYPE(ANS), the value *p* in BIF(*x*,*y*,*p*) and in BIF(*x*,*y*,*p*,0) is interpreted as specifying a binary number of digits, the operation is performed as a binary operation, and the result has the attributes FIXED BIN(*p*,0).

However, for BIF(*x*,*y*,*p*,*q*) if *q* is not zero, the operation will be performed as a decimal operation, and the result will have the attributes FIXED DEC(*t*,*u*) where *t* and *u* are the decimal equivalents of *p* and *q*, namely *t* = 1 + ceil(*p* / 3.32) and *u* = ceil(*q* / 3.32). In this case, *x*, *y*, *p*, and *q* are effectively all converted to decimal (in contrast to the DECDIGIT suboption, which converts only *x* and *y* to decimal and does so even if *q* is zero). The compiler will issue the informational message 1BM1053 in this situation.

DECDIGIT

Under PRECTYPE(DECDIGIT), the value *p* in BIF(*x*,*y*,*p*) and BIF(*x*,*y*, *p*,0) is interpreted as specifying a decimal number of digits, the operation is performed as a binary operation, and the result has the attributes FIXED BIN(*s*) where *s* is the corresponding binary equivalent to *p* (namely *s* = ceil(3.32*p*)). For an instance of BIF(*x*,*y*,*p*, *q*) where *q* is not zero, the results under PRECTYPE(DECDIGIT) are the same as the results under PRECTYPE(DECRESULT).

DECRESULT

Under PRECTYPE(DECRESULT), the value *p* in BIF(*x*,*y*,*p*) and the values *p* and *q* in BIF(*x*,*y*,*p*,*q*) are interpreted as specifying a decimal number of digits, the operation is performed as a decimal operation, and the result has the attributes FIXED DEC(*p*,0) or FIXED DEC(*p*,*q*) respectively. The result is the same as would be produced if the DECIMAL built-in were applied to *x* and *y*.

PREFIX

The PREFIX option enables or disables the specified PL/I conditions in the compilation unit being compiled without you having to change the source program. The specified condition prefixes are logically prefixed to the beginning of the first PACKAGE or PROCEDURE statement.

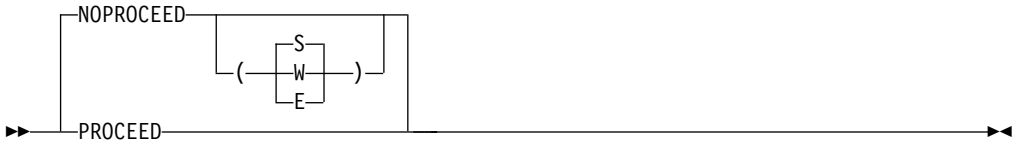
condition
Any condition that can be enabled or disabled in a PL/I program, as explained in the *PL/I Language Reference*.

The use of the PREFIX option with one or more of the checkout conditions (SIZE, STRINGRANGE, STRINGSIZE, and SUBSCRIPTRANGE) can significantly increase the time and space needed for a compilation.

Default: PREFIX(CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE UNDERFLOW ZERODIVIDE)

PROCEED

The PROCEED option stops the compiler after processing by a preprocessor is completed depending on the severity of messages issued by previous preprocessors.



ABBREVIATIONS: PRO, NPRO

PROCEED
Is equivalent to NOPROCEED(S).

NOPROCEED
Ends the processing after the preprocessor has finished compiling.

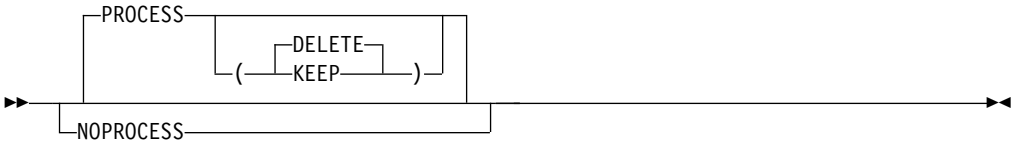
NOPROCEED(S)
The invocation of preprocessors and the compiler does not continue if a severe or an unrecoverable error is detected in this stage of preprocessing.

NOPROCEED(E)
The invocation of preprocessors and the compiler does not continue if an error, a severe error, or an unrecoverable error is detected in this stage of preprocessing.

NOPROCEED(W)
The invocation of preprocessors and the compiler does not continue if a warning, an error, a severe error, or an unrecoverable error is detected in this stage of preprocessing.

PROCESS

The PROCESS option determines whether *PROCESS statements are allowed and, if they are allowed, whether they are written to the MDECK file.



Under the NOPROCESS option, the compiler will flag any *PROCESS statement with an E-level message.

Under the PROCESS(KEEP) option, the compiler will not flag *PROCESS statements, and the compiler will retain any *PROCESS statements in the MDECK output.

Under the PROCESS(DELETE) option, the compiler will not flag *PROCESS statements, but the compiler will not retain any *PROCESS statements in the MDECK output.

QUOTE

The QUOTE option specifies an alternate symbol that can be used as the quote character.

Note: Do not code any blanks between the quotation marks.

The IBM-supplied default code point for the QUOTE symbol is "".

char
A single SBCS character

You cannot specify any of the alphabetic characters, digits, and special characters defined in the *PL/I Language Reference*, except for the standard QUOTE symbol (").

You must specify a valid character.

The QUOTE option is ignored if the GRAPHIC option is also specified.

REDUCE

The REDUCE option specifies that the compiler is permitted to reduce an assignment of a null string to a structure into simpler operations - even if that means padding bytes might be overwritten.

The REDUCE option also allows the compiler to reduce the assignment of matching structures into a simple aggregate move - even if the structures contain POINTER fields.

The NOREDUCE option specifies that the compiler must decompose an assignment of a null string to a structure into a series of assignments of the null string to the base members of the structure.

Under the NOREDUCE option, BY NAME assignments that can be reduced to aggregate moves are not reduced if the elements that would be moved together have the AREA or VARYING(Z) attributes.

The REDUCE option causes fewer lines of code to be generated for an assignment of a null string to a structure, and that usually means your compilation is quicker and your code runs much faster. However, padding bytes might be zeroed out.

For instance, in the following structure, there is one byte of padding between *field12* and *field13*.

```
dcl
 1 sample ext,
   5  field10          bin fixed(31),
   5  field11          bin fixed(15),
   5  field12          bit(8),
   5  field13          bin fixed(31);
```

Now consider the assignment `sample = '';`.

Under the NOREDUCE option, it will cause four assignments to be generated, and the padding byte will be unchanged.

However, under REDUCE, the assignment will be reduced to one operation, but the padding byte will be zeroed out.

The NOREDUCE option makes the compiler act more like the OS PL/I and the PL/I for MVS compilers. These compilers would reduce an assignment of matching structures into a simple aggregate move unless the structures contain POINTER fields. The NOREDUCE option will make this compiler act the same way.

RENT

Your code is "naturally reentrant" if it does not alter any of its static variables. The RENT option specifies that the compiler is to take code that is not naturally reentrant and make it reentrant.

For a detailed description of reentrancy, see the *z/OS Language Environment Programming Guide*. If you use the RENT option, the Linkage Editor cannot directly process the object module that is produced; you must use PDSEs.

Note: Under the LP(64) option, the RENT option is ignored; effectively, RENT is always on.

The NORENT option specifies that the compiler is not to specifically generate reentrant code from nonreentrant code. Any naturally reentrant code remains reentrant.

If you link a module (either MAIN or FETCHABLE) containing one or more programs compiled with the RENT option, you must specify DYNAM=DLL and REUS=RENT on the link step.

If you specify the options NORENT and LIMITS(EXTNAME(*n*)) (with *n* <= 7), the text decks generated by the compiler will have the same format as those generated by the older PL/I compilers. If you use any other options, you must use PDSEs.

The code generated under the NORENT option might not be reentrant unless the NOWRITABLE option is also specified.

The use of the NORENT does preclude the use of some features of the compiler. In particular, note the following considerations:

- DLLs cannot be built.
- Reentrant, writeable static is not supported.
- A STATIC ENTRY VARIABLE cannot have an INITIAL value.

You can mix RENT and NORENT code subject to the following restrictions:

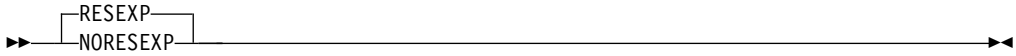
- Code compiled with RENT cannot be mixed with code compiled with NORENT if they share any EXTERNAL STATIC variables.
- Code compiled with RENT cannot call an ENTRY VARIABLE set in code compiled with NORENT.
- Code compiled with RENT cannot call an ENTRY CONSTANT that was fetched in code compiled with NORENT.
- Code compiled with RENT can fetch a module containing code compiled with NORENT if one of the following conditions is true:
  - All the code in the fetched module was compiled with NORENT.
  - The code containing the entry point to the module was compiled with RENT.
- Code compiled with NORENT code cannot fetch a module containing any code compiled with RENT.
- Code compiled with NORENT WRITABLE cannot be mixed with code compiled with NORENT NOWRITABLE if they share any external CONTROLLED variables or any external FILEs.

Given the above restrictions, the following is still valid:

- A NORENT routine, called say `mnorent`, statically links and calls a RENT routine, called say `mrent`.
- The RENT routine `mrent` then fetches and calls a separately-linked module with an entry point compiled with RENT.

RESEXP

The RESEXP option specifies that the compiler is permitted to evaluate all restricted expressions at compile time even if this would cause a condition to be raised and the compilation to end with S-level messages.



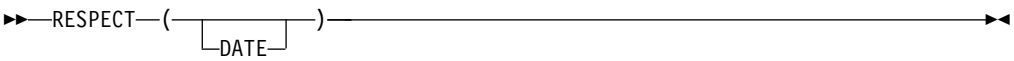
Under the NORESEXP compiler option, the compiler will still evaluate all restricted expression occurring in declarations, including those in INITIAL value clauses.

For example, under the NORESEXP option, the compiler will not flag the following statement (and the ZERODIVIDE exception will be raised at run time).

```
if preconditions_not_met then
   x = 1 / 0;
```

RESPECT

The RESPECT option causes the compiler to honor any specification of the DATE attribute and to apply the DATE attribute to the result of the DATE built-in function.



Using the default, `RESPECT()`, causes the compiler to ignore any specification of the DATE attribute and ensures that the compiler does not apply the DATE attribute to the result of the DATE built-in function.

RTCHECK

The RTCHECK option specifies that extra code is generated to force the ERROR condition to be raised if a null pointer is dereferenced, that is, if the pointer is used to change or obtain the value of a variable.

NULLPTR
Extra code is generated to force the ERROR condition to be raised if a null pointer, that is, a pointer equal to SYSNULL(), is dereferenced. To use NULLPTR, you must specify ARCH(8).

NULL370
Extra code is generated to force the ERROR condition to be raised if a pointer equal to the old NULL() value is dereferenced. The old NULL() value is the hexadecimal value of 'FF000000'x. To use NULL370, you must specify ARCH(8) or greater.

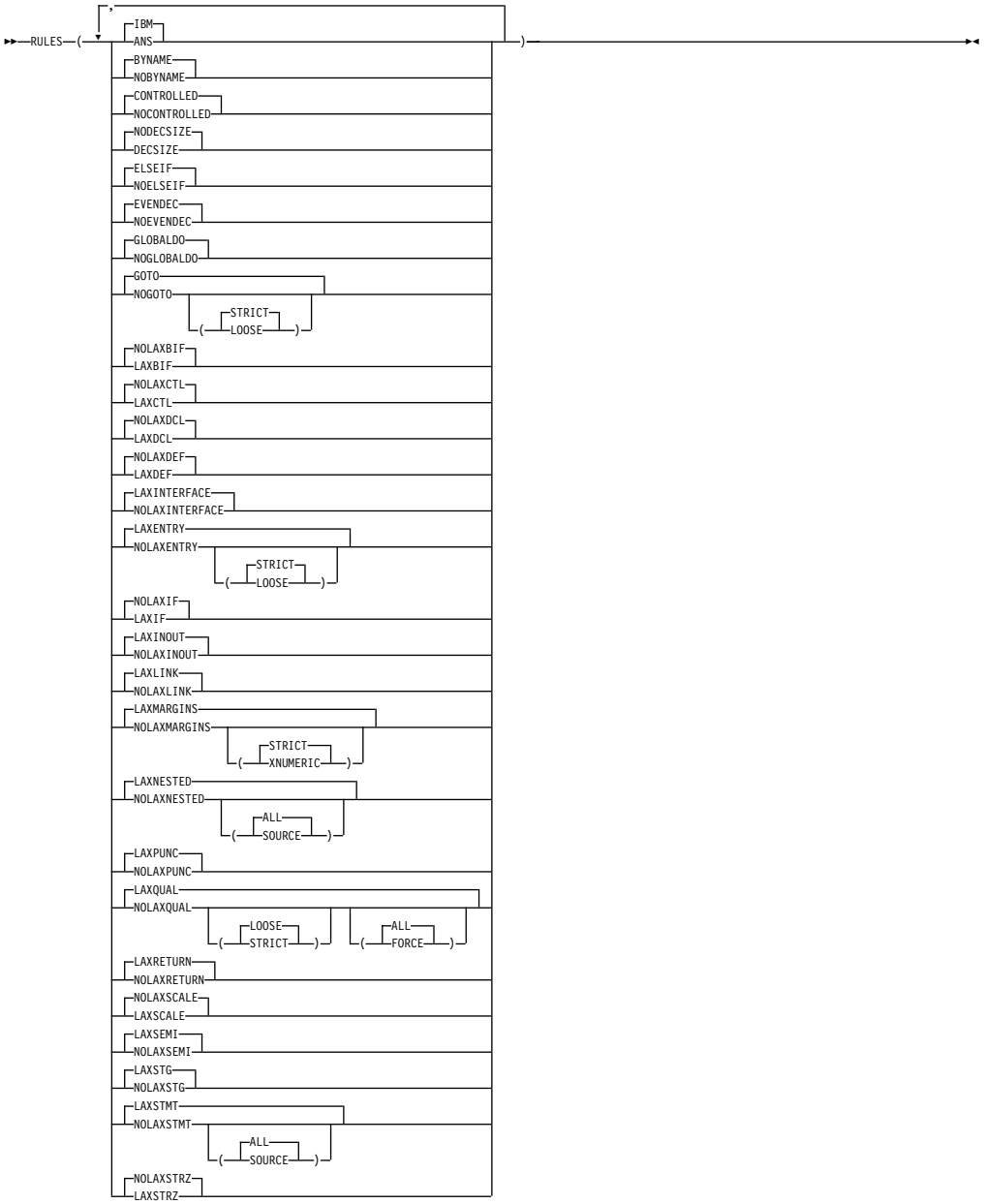
NONULLPTR
No extra code is generated to force the ERROR condition to be raised if a null pointer is dereferenced.

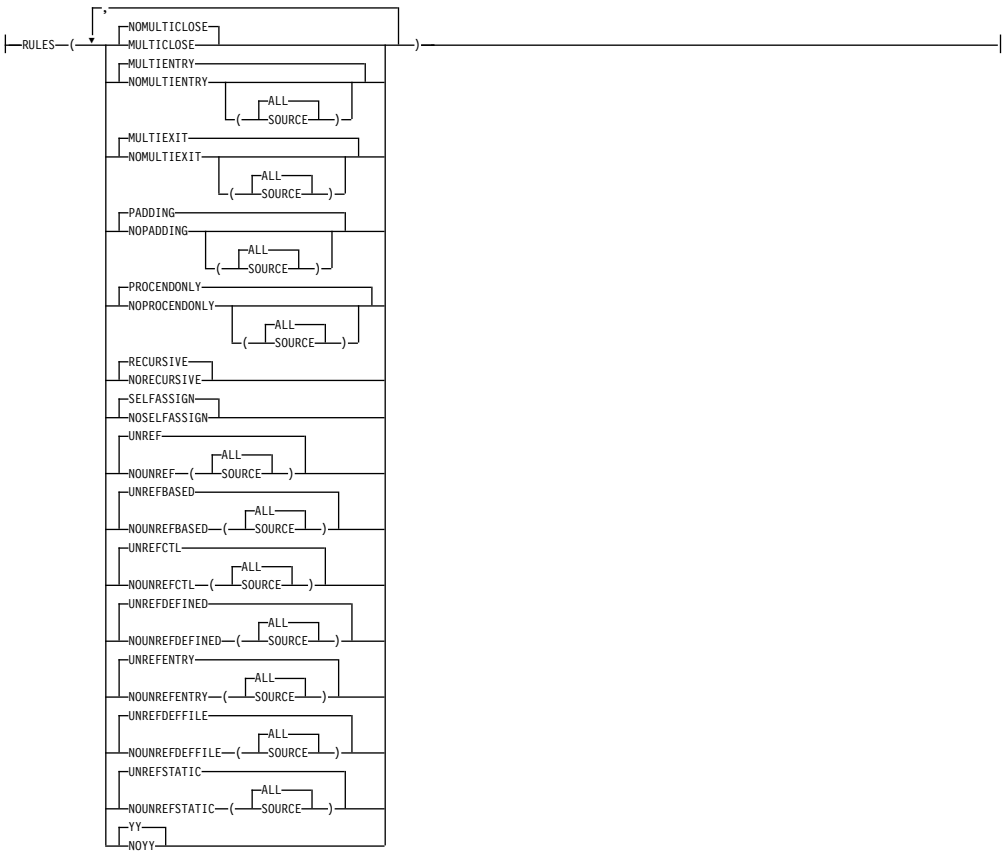
The default is RTCHECK(NONULLPTR).

Note: When a null pointer is dereferenced, a compare-and-trap data exception occurs.

RULES

The RULES option allows or disallows certain language capabilities and lets you choose semantics when alternatives are available. It can help you diagnose common programming errors.


RULES
IBM
ANS
BYNAME
NOBYNAME
CONTROLLED
NOCONTROLLED
NODECSIZE
DECSIZE
ELSEIF
NOELSEIF
EVENDEC
NOEVENDEC
GLOBALDO
NOGLOBALDO
GOTO
NOGOTO
STRICT
LOOSE
NOLAXBIF
LAXBIF
NOLAXCTL
LAXCTL
NOLAXDCL
LAXDCL
NOLAXDEF
LAXDEF
LAXINTERFACE
NOLAXINTERFACE
LAXENTRY
NOLAXENTRY
STRICT
LOOSE
NOLAXIF
LAXIF
LAXINOUT
NOLAXINOUT
LAXLINK
NOLAXLINK
LAXMARGINS
NOLAXMARGINS
STRICT
XNUMERIC
LAXNESTED
NOLAXNESTED
ALL
SOURCE
LAXPUNC
NOLAXPUNC
LAXQUAL
NOLAXQUAL
LOOSE
STRICT
ALL
FORCE
LAXRETURN
NOLAXRETURN
NOLAXSCALE
LAXSCALE
LAXSEMI
NOLAXSEMI
LAXSTG
NOLAXSTG
LAXSTMT
NOLAXSTMT
ALL
SOURCE
NOLAXSTRZ
LAXSTRZ



IBM | ANS

Under the IBM suboption:

- For operations requiring string data, data with the BINARY attribute is converted to BIT.
- Conversions in arithmetic operations or comparisons occur as described in the *PL/I Language Reference*.
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *PL/I Language Reference* except that operations specified as scaled fixed binary are evaluated as scaled fixed decimal.
- Nonzero scale factors are permitted in FIXED BIN declarations.
- If the result of any precision-handling built-in function (ADD, BINARY, and so on) has FIXED BIN attributes, the specified or implied scale factor can be nonzero.
- Even if all arguments to the MAX or MIN built-in functions are UNSIGNED FIXED BIN, all the arguments are converted to SIGNED, the function is evaluated with these converted arguments, and the result is always SIGNED.
- Even when you add, multiply, or divide two UNSIGNED FIXED BIN operands, all the operands are converted to SIGNED, the operation is evaluated with these converted arguments, and the result has the SIGNED attribute.
- Even when you apply the MOD or REM built-in functions to two UNSIGNED FIXED BIN operands, all the arguments are converted to SIGNED, the function is evaluated with these converted arguments, and the result has the SIGNED attribute.

- Declaring a variable with the OPTIONS attribute implies the ENTRY attribute.

Under the ANS suboption:

- For operations requiring string data, data with the BINARY attribute is converted to CHARACTER.
- Conversions in arithmetic operations or comparisons occur as described in the *PL/I Language Reference*.
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *PL/I Language Reference*.
- Nonzero scale factors are **not** permitted in FIXED BIN declares.
- If the result of any precision-handling built-in function (ADD, BINARY, and so on) has FIXED BIN attributes, the specified or implied scale factor must be zero.
- If all arguments to the MAX or MIN built-in functions are UNSIGNED FIXED BIN, the result is UNSIGNED.
- When you add, multiply, or divide two UNSIGNED FIXED BIN operands, the result has the UNSIGNED attribute.
- When you apply the MOD or REM built-in functions to two UNSIGNED FIXED BIN operands, the result has the UNSIGNED attribute.
- Declaring a variable with the OPTIONS attribute does not imply the ENTRY attribute.

Also, under RULES(ANS), the following errors, which the old compilers ignored, produce E-level messages:

- Specifying a string constant as the argument to the STRING built-in
- Giving too many asterisks as subscripts in an array reference
- Qualifying a CONTROLLED variable with a POINTER reference (as if the CONTROLLED variable were BASED)

The default is RULES(IBM).

BYNAME | NOBYNAME

Specifying NOBYNAME causes the compiler to flag all BYNAME assignments with an E-level message.

The default is RULES(BYNAME).

CONTROLLED | NOCONTROLLED

Specifying NOCONTROLLED causes the compiler to flag any use of the CONTROLLED attribute.

Specifying CONTROLLED causes the compiler not to flag the use of the CONTROLLED attribute.

The default is RULES(CONTROLLED).

DECSIZE | NODECSIZE

Specifying DECSIZE causes the compiler to flag any assignment of a FIXED DECIMAL expression to a FIXED DECIMAL variable when the SIZE condition is disabled if the SIZE condition could be raised by the assignment.

Specifying RULES(DECSIZE) might cause the compiler to produce many messages because when SIZE is disabled, any statement of the form X = X + 1 will be flagged if X is FIXED DECIMAL.

The default is RULES(NODECSIZE).

ELSEIF | NOELSEIF
Specifying NOELSEIF causes the compiler to flag any ELSE statement that is immediately followed by an IF statement and suggest that it be rewritten as a SELECT statement.

This option can be useful in enforcing that SELECT statements be used rather than a series of nested IF-THEN-ELSE statements.

The RULES(NOELSEIF) compiler option now also flag ELSE statements immediately followed by an IF statement that is enclosed in a simple DO-END.

The default is RULES(ELSEIF).

EVENDEC | NOEVENDEC
Specifying NOEVENDEC causes the compiler to flag any FIXED DECIMAL declaration that specifies an even precision.

The default is RULES(EVENDEC).

GLOBALDO | NOGLOBALDO
Specifying NOGLOBALDO instructs the compiler to flag all DO loops with control variables that are declared in a parent block.

The default is RULES(GLOBALDO).

GOTO | NOGOTO
Specifying NOGOTO(LOOSE) causes the compiler to flag any GOTO statement to a label constant unless the GOTO is exiting an ON-unit or unless the target label constant is in the same block as the GOTO statement.

Specifying NOGOTO(STRICT) causes the compiler to flag any GOTO statement to a label constant unless the GOTO is exiting an ON-unit.

The default is RULES(GO). When you specifying RULES(NOGOTO), the default is STRICT.

LAXBIF | NOLAXBIF
Specifying LAXBIF causes the compiler to build a contextual declaration for built-in functions, such as NULL, even when used without an empty parameter list.

The default is RULES(NOLAXBIF).

LAXCTL | NOLAXCTL
Specifying LAXCTL allows a CONTROLLED variable to be declared with a constant extent and yet to be allocated with a differing extent. NOLAXCTL requires that if a CONTROLLED variable is to be allocated with a varying extent, that extent must be specified as an asterisk or as a non-constant expression.

The following code is illegal under NOLAXCTL:

```
dcl a bit(8) ctl;
alloc a;
alloc a bit(16);
```

But this code would still be valid under NOLAXCTL:

```
dcl b bit(n) ctl;
dcl n fixed bin(31) init(8);
alloc b;
alloc b bit(16);
```

The default is RULES(NOLAXCTL).

LAXDCL | NOLAXDCL
Specifying LAXDCL allows implicit declarations. NOLAXDCL disallows all implicit and contextual declarations except for BUILTINs and for files SYSIN and SYSPRINT.

The default is RULES(NOLAXDCL).

LAXDEF | NOLAXDEF
Specifying LAXDEF allows so-called illegal defining to be accepted without any compiler messages (rather than the E-level messages that the compiler would usually produce).

The default is RULES(NOLAXDEF).

LAXENTRY | NOLAXENTRY
Specifying LAXENTRY allows unprototyped entry declarations. Specifying NOLAXENTRY causes the compiler to flag all unprototyped entry declarations, that is, all ENTRY declares that do not specify a parameter list.

Note: If an ENTRY should have no parameters, it should be declared as ENTRY() rather than simply as ENTRY.

STRICT
Specifying RULES(NOLAXENTRY(STRICT)) causes the compiler to flag unprototyped entry declarations that have the OPTIONS(ASM) attribute.

LOOSE
Specifying RULES(NOLAXENTRY(LOOSE)) causes the compiler not to flag unprototyped entry declarations that have the OPTIONS(ASM) attribute.

The default is RULES(LAXENTRY). When you specifying RULES(NOLAXENTRY), the default is STRICT.

LAXIF | NOLAXIF
Specifying RULES(NOLAXIF) causes the compiler to flag any IF, WHILE, UNTIL, and WHEN clauses that do not have the attributes BIT(1) NONVARYING. It also causes the compiler to flag the assignments of the form `x=y=z`, but it does not flag assignments of the `x=(y=z)` form.

The following code will all be flagged under NOLAXIF:

```
dcl i fixed bin;
dcl b bit(8);
   .
   .
   .
if i then ...
if b then ...
```

The default is RULES(NOLAXIF).

LAXINOUT | NOLAXINOUT
Specifying NOLAXINOUT causes the compiler to assume that all ASSIGNABLE BYADDR parameters are input (and possibly output) parameters and hence to issue a warning if the compiler thinks such a parameter has not been initialized.

The default is RULES(LAXINOUT).

LAXINTERFACE | NOLAXINTERFACE
Specifying NOLAXINTERFACE causes the compiler to flag code that does not contain a valid explicit declare for each of its external PROCEDUREs.

- Under NOLAXINTFACE, the compiler will not flag any MAIN routines.

- Under NOLAXINTERFACE, the compiler will flag any PROCEDURE that does not have an explicit declare at the PACKAGE level (and hence any code not in a PACKAGE).
- Under NOLAXINTERFACE, the compiler will also flag any PROCEDUREs for which the implicit and explicit declarations do not match.

The default is RULES(LAXINTERFACE).

LAXLINK | NOLAXLINK

Specifying NOLAXLINK causes the compiler to flag any assign or compare of two ENTRY variables or constants if any of the following do not match:

- The parameter description lists

  For instance, if A1 is declared as ENTRY(CHAR(8)) and A2 as ENTRY(POINTER) VARIABLE, under RULES(NOLAXLINK) the compiler will flag an attempt to assign A1 to A2.
- The RETURNS attribute

  For instance, if A3 is declared as ENTRY RETURNS(FIXED BIN(31)) and A4 as an ENTRY VARIABLE without the RETURNS attribute, under RULES(NOLAXLINK) the compiler will flag an attempt to assign A3 to A4.
- The LINKAGE and other OPTIONS suboptions

  For instance, if A5 is declared as ENTRY OPTIONS(ASM) and A6 as an ENTRY VARIABLE without the OPTIONS attribute, under RULES(NOLAXLINK) the compiler will flag an attempt to assign A5 to A6. This is because the OPTIONS(ASM) in the declare of A5 implies that A5 has LINKAGE(SYSTEM)), and in contrast, because A6 has no OPTIONS attribute, it will have LINKAGE(OPTLINK) by default).

The default is RULES(LAXLINK).

LAXMARGINS | NOLAXMARGINS

Specifying NOLAXMARGINS causes the compiler to flag, depending on the setting of the STRICT and XNUMERIC suboption, lines containing nonblank characters after the right margin. This can be useful in detecting code, such as a closing comment, that has accidentally been pushed out into the right margin.

If the NOLAXMARGINS and STMT options are used together with one of the preprocessors, any statements that would be flagged because of the NOLAXMARGINS option will be reported as statement zero (because statement numbering occurs only after all the preprocessors are finished, but the detection of text outside the margins occurs as soon as the source is read).

STRICT

Under STRICT, the compiler flags any line that contains nonblank characters after the right margin.

XNUMERIC

Under XNUMERIC, the compiler flags any line that contains nonblank characters after the right margin except if the right margin is column 72 and columns 73 through 80 all contain numeric digits.

The default is RULES(LAXMARGINS). When you specifying RULES(NOLAXMARGINS), the default is ALL.

LAXNESTED | NOLAXNESTED

Specifying RULES(LAXNESTED) causes the compiler not to flag the executable code in a procedure that follows any subprocedures.

Specifying RULES(NOLAXNESTED) causes the compiler to flag any executable code in a procedure that follows any subprocedures.

ALL
Under ALL, all violations of RULES(NOLAXNESTED) are flagged. ALL is the default.

SOURCE
Under SOURCE, only those violations that occur in the primary source file are flagged.

The default is RULES(LAXNESTED). When you specifying RULES(NOLAXNESTED), the default is ALL.

LAXPUNC | NOLAXPUNC
Specifying NOLAXPUNC causes the compiler to flag with an E-level message any place where it assumes punctuation is missing.

For instance, given the statement `I = (1 * (2);`, the compiler assumes that a closing right parenthesis is meant before the semicolon. Under RULES(NOLAXPUNC), this statement will be flagged with an E-level message; otherwise, it will be flagged with a W-level message.

The default is RULES(LAXPUNC).

LAXQUAL | NOLAXQUAL
Specifying NOLAXQUAL(LOOSE) causes the compiler to flag any reference to structure members that are not level 1 and are not dot qualified. Consider the following example:

```
dcl
  1 a,
    2 b,
      3 b fixed bin,
      3 c fixed bin;

c   = 11;   /* would be flagged */
b.c = 13;   /* would not be flagged */
a.c = 17;   /* would not be flagged */
```

Specifying NOLAXQUAL(STRICT) causes the compiler to flag any reference to structure members that do not include the level-1 name. Consider the following example:

```
dcl
  1 a,
    2 b,
      3 b fixed bin,
      3 c fixed bin;

c   = 11;   /* would be flagged */
b.c = 13;   /* would be flagged */
a.c = 17;   /* would not be flagged */
```

ALL
Under ALL, all violations of RULES(NOLAXQUAL) are flagged. ALL is the default.

FORCE
Under FORCE, only those violations that occur in structures with the FORCE(NOLAXQUAL) attribute are flagged.

The default is RULES(LAXQUAL). When you specifying RULES(NOLAXQUAL), LOOSE and ALL are defaults.

LAXRETURN | NOLAXRETURN

Specifying NOLAXRETURN causes the compiler to generate code to raise the ERROR condition when a RETURN statement is used in either of the following ways:

- With an expression in a procedure that is coded without the RETURNS option
- Without an expression in a procedure that is coded with the RETURNS option

ERROR will also be raised if the code falls through to the END statement in a PROCEDURE with the RETURNS attribute.

The default is RULES(LAXRETURN).

LAXSCALE | NOLAXSCALE

Specifying NOLAXSCALE causes the compiler to flag any FIXED BIN(p,q) or FIXED DEC(p,q) declaration where $q < 0$ or $p < q$.

It also causes the compiler to flag ROUND(x,p) when $p < 0$.

The message issued when the compiler flags ROUND(x,p) is different from that issued when the compiler flags the FIXED BIN(p,q) or FIXED DEC(p,q) declaration. Therefore, you can use the EXIT option to suppress the message issued when ROUND(x,p) is flagged and keep the message for other questionable declarations.

The default is RULES(NOLAXSCALE).

LAXSEMI | NOLAXSEMI

Specifying NOLAXSEMI causes the compiler to flag any semicolons appearing inside comments.

The default is RULES(LAXSEMI).

LAXSTG | NOLAXSTG

Specifying NOLAXSTG causes the compiler to flag declarations where a variable A is declared as BASED on ADDR(B) and STG(A) > STG(B) even (and this is the key part) if B is a parameter.

Note that even with NOLAXSTG specified, if B has subscripts, no IBM2402I E-level message will be produced.

The compiler would already flag this kind of problem if B were in AUTOMATIC or STATIC storage, but it does not, by default, flag this when B is a parameter (because some programmers declare B with placeholder attributes that do not describe the actual argument). For situations where parameter and argument declarations match (or should match), specifying RULES(NOLAXSTG) can help detect more storage overlay problems.

The default is RULES(LAXSTG).

LAXSTMT | NOLAXSTMT

Specifying NOLAXSTMT causes the compiler to flag any line that has more than one statement.

ALL

Specifying RULES(NOLAXSTMT(ALL)) causes the compiler to flag all violations of NOLAXSTMT. ALL is the default.

SOURCE

Specifying RULES(NOLAXSTMT(SOURCE)) causes the compiler to flag only those violations in the primary source file.

Additionally, NOLAXSTMT accepts EXCEPT with a (possibly empty) list of keywords that are not be flagged when a second statement on a line begins with one of these keywords. For example, this can allow `DO;` to appear on the same line as an `IF ... THEN` statement.

The following keywords are allowed in EXCEPT:

allocate
assert
attach
begin
call
cancel
close
declare
define
delay
delete
detach
display
do
else
end
exit
fetch
flush
free
get
go
goto
halt
if
iterate
leave
locate
on
open
otherwise
put
read
reinit
release
resignal
return
revert
rewrite
select
signal
stop
unlock
wait
when
write

The default is RULES(LAXSTMT). When you specifying RULES(NOLAXSTMT), the default is ALL.

LAXSTRZ | NOLAXSTRZ
Specifying LAXSTRZ causes the compiler not to flag any bit or character variable that is initialized to or assigned a constant value that is too long if the excess bits are all zeros (or if the excess characters are all blank).

The default is RULES(NOLAXSTRZ).

MULTICLOSE | NOMULTICLOSE
Specifying NOMULTICLOSE causes the compiler to flag all statements that force the closure of multiple groups of statement with an E-level message.

The default is RULES(NOMULTICLOSE).

MULTIENTRY | NOMULTIENTRY
Specifying NOMULTIENTRY causes the compiler to flag code that contains ENTRY statements.

ALL
Specifying NOMULTIENTRY(ALL) causes the compiler to flag all violations of RULES(NOMULTIENTRY).

SOURCE
Specifying NOMULTIENTRY(SOURCE) causes the compiler to flag only those violations that occur in the primary source file.

The default is RULES(MULTIENTRY). When you specifying RULES(NOMULTIENTRY), the default sub-suboption is ALL.

MULTIEXIT | NOMULTIEXIT
Specifying NOMULTIEXIT causes the compiler to flag code that contains multiple RETURN statements.

ALL
Specifying NOMULTIEXIT(ALL) causes the compiler to flag all violations of RULES(NOMULTIEXIT).

SOURCE
Specifying NOMULTIEXIT(SOURCE) causes the compiler to flag only those violations that occur in the primary source file.

The default is RULES(MULTIEXIT). When you specifying RULES(NOMULTIEXIT), the default sub-suboption is ALL.

PADDING | NOPADDING
Specifying NOPADDING causes the compiler to flag all structures that contain padding.

ALL
Specifying NOPADDING(ALL) causes the compiler to flag all violations of RULES(NOPADDING).

SOURCE
Specifying NOPADDING(SOURCE) causes the compiler to flag only those violations that occur in the primary source files.

The default is RULES(PADDING).

PROCENDONLY | NOPROCENDONLY
Specifying NOPROCENDONLY causes any END statement that closes a PROCEDURE to be flagged if the END statement does not name the PROCEDURE, that is, if the END keyword is immediately followed by a semicolon.

ALL
Under RULES(NOPROCENDONLY(ALL)), the compiler flags all violations of NOPROCENDONLY. ALL is the default.

SOURCE
Under RULES(NOPROCENDONLY(SOURCE)), the compiler flags only those violations in the primary source file.

The default is RULES(PROCENDONLY).

RECURSIVE | NORECURSIVE
Specifying NORECURSIVE causes the compiler to flag any use of the RECURSIVE attribute or any procedure that directly calls itself.

Specifying RECURSIVE causes the compiler not to flag the use of the RECURSIVE attribute or any procedure that directly calls itself.

Note: Do not use RULES(NORECURSIVE) and DFT(RECURSIVE) together.

The default is RULES(RECURSIVE).

SELFASSIGN | NOSELFASSIGN
Specifying NOSELFASSIGN causes the compiler to flag all assignments where the source and the target are the same.

The default is RULES(SELFASSIGN).

UNREF | NOUNREF

Specifying NOUNREF causes the compiler to flag any level-1 AUTOMATIC variable that is not referenced and that, if it is a structure or union, contains no subelement that is referenced. NOUNREF ignores variables with names that start with any of the following prefixes: DSN, DFH, EYU, and SQL.

ALL

Specifying RULES(NOUNREF(ALL)) causes the compiler to flag all unreferenced variables. When NOUNREF is specified, ALL is the default.

SOURCE

Specifying RULES(NOUNREF(SOURCE)) causes the compiler to flag unreferenced variables that are not declared in an INCLUDE file.

The default is RULES(UNREF).

UNREFBASED | NOUNREFBASED

Specfiying NOUNREFBASED suboption causes the compiler to flag unreferenced BASED variables that are in BASED storage.

ALL

Specifying RULES(NOUNREFBASED(ALL)) causes the compiler to flag all unreferenced BASED variables.

SOURCE

Specifying RULES(NOUNREFBASED(SOURCE)) causes the compiler to flag unreferenced BASED variables that are not declared in an INCLUDE file.

The default is RULES(UNREFBASED). When NOUNREFBASED is specified, ALL is the default.

UNREFCTL | NOUNREFCTL

Specfiying NOUNREFCTL suboption causes the compiler to flag unreferenced CTL variables.

ALL

Specifying RULES(NOUNREFCTL(ALL)) causes the compiler to flag all unreferenced CTL variables.

SOURCE

Specifying RULES(NOUNREFCTL(SOURCE)) causes the compiler to flag unreferenced CTL variables that are not declared in an INCLUDE file.

The default is RULES(UNREFCTL). When NOUNREFCTL is specified, ALL is the default.

UNREFDEFINED | NOUNREFDEFINED

Specfiying NOUNREFDEFINED suboption causes the compiler to flag unreferenced DEFINED variables.

ALL

Specifying RULES(NOUNREFDEFINED(ALL)) causes the compiler to flag all unreferenced DEFINED variables.

SOURCE

Specifying RULES(NOUNREFDEFINED(SOURCE)) causes the compiler to flag unreferenced DEFINED variables that are not declared in an INCLUDE file.

The default is RULES(UNREFDEFINED). When NOUNREFCTL is specified, ALL is the default.

UNREFENTRY | NOUNREFENTRY
Specfiying NOUNREFENTRY suboption causes the compiler to flag unreferenced ENTRY constants.

ALL
Specifying RULES(NOUNREFENTRY(ALL)) causes the compiler to flag all unreferenced ENTRY constants.

SOURCE
Specifying RULES(NOUNREFENTRY(SOURCE)) causes the compiler to flag unreferenced ENTRY constants that are not declared in an INCLUDE file.

The default is RULES(UNREFENTRY). When NOUNREFENTRY is specified, ALL is the default.

UNREFFILE | NOUNREFFILE
Specfiying NOUNREFFILE suboption causes the compiler to flag unreferenced FILE constants.

ALL
Specifying RULES(NOUNREFFILE(ALL)) causes the compiler to flag all unreferenced FILE constants.

SOURCE
Specifying RULES(NOUNREFFILE(SOURCE)) causes the compiler to flag unreferenced FILE constants that are not declared in an INCLUDE file.

The default is RULES(UNREFFILE). When NOUNREFFILE is specified, ALL is the default.

UNREFSTATIC | NOUNREFSTATIC
Specfiying NOUNREFSTATIC suboption causes the compiler to flag unreferenced STATIC variables.

ALL
Specifying RULES(NOUNREFSTATIC(ALL)) causes the compiler to flag all unreferenced STATIC variables.

SOURCE
Specifying RULES(NOUNREFSTATIC(SOURCE)) causes the compiler to flag unreferenced STATIC variables that are not declared in an INCLUDE file.

The default is RULES(UNREFSTATIC). When NOUNREFSTATIC is specified, ALL is the default.

YY | NOYY
Specifying NOYY suboption causes the compiler to flag the use of 2-digit years, including:

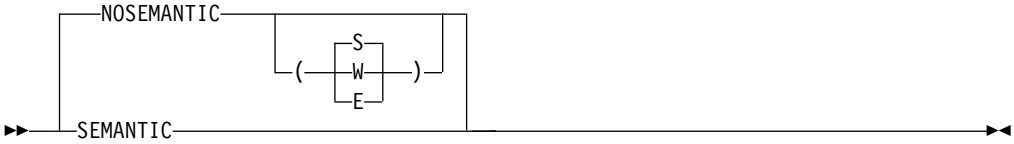
- date-time patterns with the year specified as YY or ZY,
- the DATE attribute without a date-time pattern (since that implies a pattern of YYMMDD),
- Y4DATE, Y4JULIAN, and Y4YEAR built-in functions,
- DATE built-in function,
- date-time functions with a window argument.

The default is RULES(YY).

Default: RULES (IBM BYNAME CONTROLLED NODECSIZE EVENDEC ELSEIF GLOBALDO GOTO NOLAXBIF NOLAXCTL NOLAXDCL NOLAXDEF LAXENTRY NOLAXIF LAXINOUT LAXINTERFACE LAXLINK LAXNESTED LAXPUNC LAXMARGINS(STRICT) LAXQUAL LAXRETURN NOLAXSCALE LAXSEMI LAXSTG LAXSTMT NOLAXSTRZ NOMULTICLOSE MULTIENTRY MULTIEXIT PADDING PROCENDONLY RECURSIVE SELFASSIGN UNREF UNREFBASED UNREFCTL UNREFDEFINED UNREFENTRY UNREFFILE UNREFSTATIC YY)

SEMANTIC

The SEMANTIC option specifies that the execution of the semantic checking stage depends on the severity of messages issued before this stage of processing.



ABBREVIATIONS: SEM, NSEM

SEMANTIC
Equivalent to NOSEMANTIC(S).

NOSEMANTIC
Processing stops after syntax checking. No semantic checking is performed.

NOSEMANTIC (S)
No semantic checking is performed if a severe error or an unrecoverable error has been encountered.

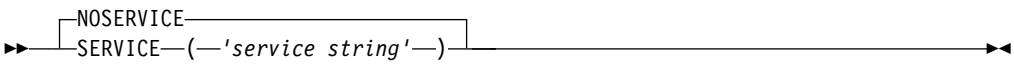
NOSEMANTIC (E)
No semantic checking is performed if an error, a severe error, or an unrecoverable error has been encountered.

NOSEMANTIC (W)
No semantic checking is performed if a warning, an error, a severe error, or an unrecoverable error has been encountered.

Semantic checking is not performed if certain kinds of severe errors are found. If the compiler cannot validate that all references resolve correctly (for example, if built-in function or entry references are found with too few arguments) the suitability of any arguments in any built-in function or entry reference is not checked.

SERVICE

The SERVICE option places a string in the object module, if generated. This string is loaded into memory with any load module into which this object is linked, and if the LE dump includes a traceback, this string will be included in that traceback.



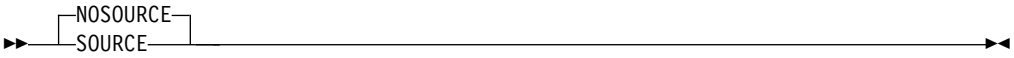
ABBREVIATIONS: SERV, NOSERV

The string is limited to 64 characters in length.

To ensure that the string remains readable across locales, only characters from the invariant character set should be used.

SOURCE

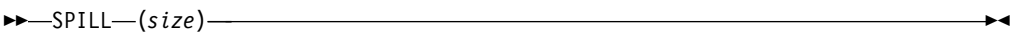
The SOURCE option specifies that the compiler includes a listing of the source program in the compiler listing. The source program listed is either the original source input or, if any preprocessors were used, the output from the last preprocessor.



ABBREVIATIONS: S, NS

SPILL

The SPILL option specifies the size of the spill area to be used for the compilation. When too many registers are in use at once, the compiler dumps some of the registers into temporary storage that is called the spill area.



ABBREVIATIONS: SP

If you have to expand the spill area, you will receive a compiler message telling you the size to which you should increase it. When you know the spill area that your source program requires, you can specify the required size (in bytes) as shown in the syntax diagram above. The maximum spill area size is 3900. Typically, you need to specify this option only when compiling very large programs with OPTIMIZE.

STATIC

The STATIC option controls whether INTERNAL STATIC variables are retained in the object module even if unreferenced.

SHORT
INTERNAL STATIC will be retained in the object module only if used.

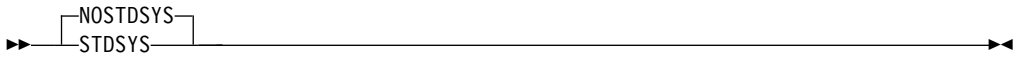
FULL
All INTERNAL STATIC with INITIAL will be retained in the object module.

If INTERNAL STATIC variables are used as "eyecatchers", you should specify the STATIC(FULL) option to ensure that they will be in the generated object module.

STDSYS

The STDSYS option specifies that the compiler should cause the SYSPRINT file to be equated to the C stdout file and the SYSIN file to be equated to the C stdin file.

Note: Under the LP(64) option, the STDSYS option is ignored; effectively, STDSYS is always on.



Using the STDSYS option might make it easier to develop and debug a mixed PL/I and C application.

When SYSPRINT is equated to stdout, its LINESIZE cannot be greater than 132 (the largest value allowed by C).

STMT

The STMT option specifies that statements in the source program are to be counted and that this statement number is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, SOURCE, and XREF options.

The default is NOSTMT.

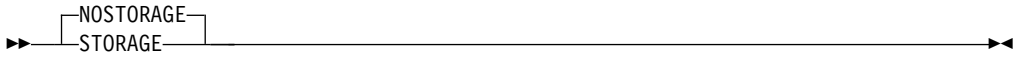
When the STMT option is specified, the source and message listings include both the logical statement numbers and the source file numbers.

Note that the GOSTMT option does not exist. The only option that produces information at run time identifying where an error has occurred is the GONUMBER option. When the GONUMBER option is used, the term *statement* in the runtime error messages refers to line numbers as used by the NUMBER compiler option even if the STMT option is in effect.

NUMBER and STMT are mutually exclusive and specifying one will negate the other.

STORAGE

The STORAGE option directs the compiler to produce as part of the listing a summary of the storage used by each procedure and begin-block.



ABBREVIATIONS: STG, NSTG

The STORAGE output also includes the amount of storage used for the internal static for the compilation.

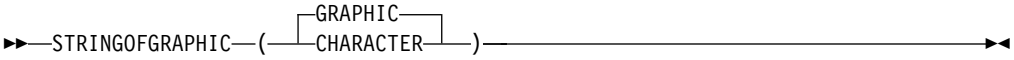
The compiler generates inline code for STG(*x*) where *x* has the BASED attribute with the REFER option and meets both of the following conditions:

- Elements in *x* with the NONVARYING and BIT attributes have the ALIGNED attribute.

- All other elements in *x* have the UNALIGNED attribute.

STRINGOFGRAPHIC

The STRINGOFGRAPHIC option determines whether the result of the STRING built-in function when applied to a GRAPHIC aggregate has the attribute CHARACTER or GRAPHIC.



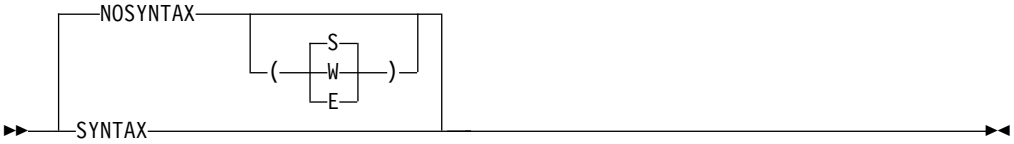
ABBREVIATIONS: CHAR, G

CHARACTER
Under STRINGOFGRAPHIC(CHAR), if the STRING built-in is applied to an array or a structure of UNALIGNED NONVARYING GRAPHIC variables, the result will have the CHARACTER attribute.

GRAPHIC
Under STRINGOFGRAPHIC(GRAPHIC), if the STRING built-in is applied to an array or a structure of GRAPHIC variables, the result will have the GRAPHIC attribute.

SYNTAX

The SYNTAX option specifies that the compiler continues into syntax checking after preprocessing when you specify the MACRO option, unless an unrecoverable error has occurred. Whether the compiler continues with the compilation depends on the severity of the error, as specified by the NOSYNTAX option.



ABBREVIATIONS: SYN, NSYN

SYNTAX
Continues syntax checking after preprocessing unless a severe error or an unrecoverable error has occurred. SYNTAX is equivalent to NOSYNTAX(S).

NOSYNTAX
Processing stops unconditionally after preprocessing.

NOSYNTAX(W)
No syntax checking if a warning, an error, a severe error, or an unrecoverable error is detected.

NOSYNTAX(E)
No syntax checking if the compiler detects an error, a severe error, or an unrecoverable error.

NOSYNTAX(S)
No syntax checking if the compiler detects a severe error or an unrecoverable error.

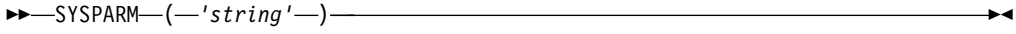
If the NOSYNTAX option terminates the compilation, no cross-reference listing, attribute listing, or other listings that follow the source program is produced.

You can use this option to prevent wasted runs when debugging a PL/I program that uses the preprocessor.

If the NOSYNTAX option is in effect, any specification of the CICS preprocessor through the CICS, XOPT or XOPTS options will be ignored. This allows the MACRO preprocessor to be invoked before the compiler invokes the CICS translator.

SYSPARM

The SYSPARM option specifies the value of the string that is returned by the macro facility built-in function SYSPARM.



string
Can be up to 64 characters long. A null string is the default.

For more information about the macro facility, see the *PL/I Language Reference*.

SYSTEM

The SYSTEM option specifies the format used to pass parameters to the MAIN PL/I procedure, and generally indicates the host system under which the program runs.

Table 5 shows the type of parameter list you can expect, and how the program runs under the specified host system. It also shows the implied settings of NOEXECOPS. Your MAIN procedure must receive only those types of parameter lists that are indicated as valid in this table. Additional runtime information for the SYSTEM option is provided in the *z/OS Language Environment Programming Guide*.

Table 5. SYSTEM option table

SYSTEM option	Type of parameter list	Program runs as	NOEXECOPS implied
SYSTEM(MVS)	Single CHARACTER string or no parameters	z/OS application program	NO
	Otherwise, arbitrary parameter list		YES
SYSTEM(CICS)	Pointer(s)	CICS transaction	YES
SYSTEM(IMS™)	Pointer(s)	IMS application program	YES
SYSTEM(OS)	z/OS UNIX parameter list	z/OS UNIX application program	YES
SYSTEM(TSO)	Pointer to CPPL[1]	TSO command processor	YES

Note:

1. See "Invoking MAIN under TSO/E" on page 183 for more details about how to invoke a MAIN procedure under TSO.

Under SYSTEM(IMS), all pointers are presumed to be passed by value (BYVALUE), but under SYSTEM(MVS) they are presumed to be passed by address (BYADDR).

MAIN procedures run under CICS must be compiled with SYSTEM(CICS) or SYSTEM(MVS).

It is highly recommended that NOEXECOPS be specified in the MAIN procedure OPTIONS option for code, such as a DB2 stored procedure, compiled with SYSTEM(MVS) but run where runtime options would not be passed.

The compiler will flag any MAIN program compiled with SYSTEM(MVS) if it has either more than one parameter or a single parameter that is not CHARACTER VARYING. It is probably better to compile such MAIN programs with SYSTEM(OS) because the library then simply passes on to MAIN the parameter list without any scanning for options or other massaging of the parameter list.

TERMINAL

The TERMINAL option determines whether diagnostic and information messages produced during compilation are displayed on the terminal.

Note: This option applies only to compilations under z/OS UNIX.

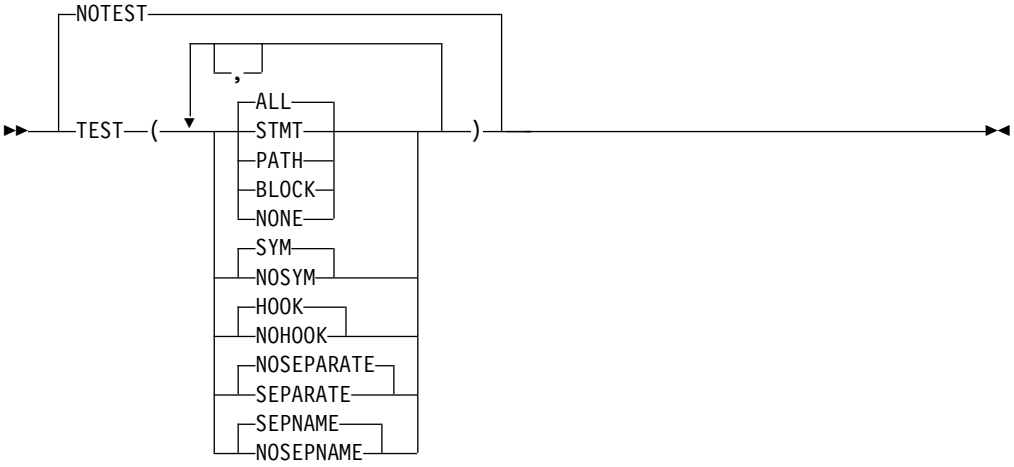
ABBREVIATIONS: TERM, NTERM

TERMINAL
Messages are displayed on the terminal.

NOTERMINAL
No information or diagnostic compiler messages are displayed on the terminal.

TEST

The TEST option specifies the level of testing capability that the compiler generates as part of the object code. You can use this option to control the location of test hooks and to control whether to generate a symbol table.



ABBREVIATIONS: AALL, ACICS, AMACRO, ASQL

STMT
Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, inserts hooks at statement boundaries and block boundaries.

PATH
Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, tells the compiler to insert hooks at these places:

- Before the first statement enclosed by an iterative DO statement
- Before the first statement of the true part of an IF statement
- Before the first statement of the false part of an IF statement
- Before the first statement of a true WHEN or OTHERWISE statement of a SELECT group
- Before the statement following a user label, excluding labeled FORMAT statements

  If a statement has multiple labels, only one hook is inserted.
- At CALLs or function references - both before and after control is passed to the routine
- At block boundaries

BLOCK
Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, tells the compiler to insert hooks at block boundaries (block entry and block exit).

ALL
Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, inserts hooks at all possible locations and generates a statement table.

Note: Under opt(2) and opt(3), hooks are set only at block boundaries.

NONE
No hooks are put into the program.

SYM
Creates a symbol table that allows you to examine variables by name.

NOSYM
No symbol table is generated.

NOTEST
Suppresses the generation of all testing information.

HOOK
Causes the compiler to insert hooks into the generated code if any of the TEST suboptions ALL, STMT, BLOCK, or PATH are in effect.

NOHOOK
Causes the compiler not to insert hooks into the generated code.

For IBM Debug Tool to generate overlay hooks, one of the suboptions ALL, PATH, STMT, or BLOCK must be specified, but HOOK need not be specified, and NOHOOK would in fact be recommended.

If NOHOOK is specified, ENTRY and EXIT breakpoints are the only PATH breakpoints at which Debug Tool will stop.

SEPARATE
Causes the compiler to place most of the debug information it generates into a separate debug file. Using this option will substantially reduce the size of the object deck created by the compiler when the TEST option is in effect.

If your program contains GET or PUT DATA statements, the separate debug file will contain less debug information because those statements require that symbol table information be placed into the object deck.

The generated debug information always includes a compressed version of the source that is passed to the compiler. This means that the source might be specified by using SYSIN DD *, or that the source might be a temporary data set that is created by an earlier job step (for example, the source might be the output of the old SQL or CICS precompilers). The suboptions that you specify for the LISTVIEW option control the content of the source.

If SEPARATE is used in a batch compilation, the JCL for that compilation must include a DD card for SYSDEBUG that must name a data set with RECFM=FB and with 80 <= LRECL <= 1024.

This suboption cannot be used with the LINEDIR compiler option.

NOSEPARATE
Causes the compiler to place all of the debug information it generates into the object deck.

Under this option, the generated debug information will not include a compressed version of the source passed to the compiler. This means that the source must in a data set that can be found by Debug Tool when you try to debug the program.

SEPNAME
Causes the compiler to place the name of the separate debug file into the object deck.

This option is ignored if the SEPARATE option is not in effect.

NOSEPNAME
Causes the compiler not to place the name of the separate debug file into the object deck.

This option is ignored if the SEPARATE option is not in effect.

Notes:

- Under LP(64), no hooks are generated and that the SEP or NOSEP compiler option is ignored.
- Under opt(2) or opt(3), hooks are set only at block boundaries. This means that debugging of optimized code is effectively limited to tracing entry and exit to PROCEDUREs and BEGIN blocks.
- You must use Debug Tool Version 6 (or later) to debug code compiled with the SEPARATE compiler option.
- There is no support for an input file that spans concatenated data sets.

Specifying TEST(NONE,NOSYM) causes the compiler to set the option to NOTEST.

Use of TEST(NONE,SYM) is strongly discouraged, and it is unclear what is intended when you specify these settings. You would probably be much better off if you specified TEST(ALL,SYM,NOHOOK) or TEST(STMT,SYM,NOHOOK).

Any TEST option other than NOTEST and TEST(NONE,NOSYM) will automatically provide the attention interrupt capability for program testing.

If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following options:
- The INTERRUPT option
- A TEST option other than NOTEST or TEST(NONE,NOSYM)

Note: ATTENTION is supported only under TSO.

The TEST option will imply GONUMBER.

Because the TEST option can increase the size of the object code and can affect performance, you might want to limit the number and placement of hooks.

If the TEST option is specified, no inlining will occur.

Structures with REFER are supported in the symbol table.

If TEST(SYM) is in effect, the compiler will generate tables to enable the automonitor feature of Debug Tool. These tables might substantially increase the size of the object module unless the TEST(SEPARATE) option is in effect. When the automonitor feature of Debug Tool is activated, these tables are used to display the values of the variables used in a statement before the statement executes - as long as the variable has computational type or has the attribute POINTER, OFFSET or HANDLE. If the statement is an assignment statement, the value of the target is also displayed; however, if the target has not been initialized or assigned previously, its value is meaningless.

Any variable declared with an * for its name is not visible when you use Debug Tool. Additionally, if an * is used as the name of a parent structure or substructure, all of its children are also invisible. Therefore, it might be better to use a single underscore for the name of any structure elements that you want to leave "unnamed".

UNROLL

The UNROLL option controls loop unrolling under optimization. Loop unrolling is an optimization that replicates a loop body multiple times and adjusts the loop control code accordingly.

AUTO
Indicates that the compiler is permitted to unroll loops that it determines are appropriate for unrolling.

Specifying the UNROLL option can increase the size of the object code that is generated.

NO Indicates that the compiler is not permitted to unroll loops.

The UNROLL option is ignored when the NOOPTIMIZE option is in effect.

Loop unrolling improves the performance of a program by exposing instruction level parallelism for instruction scheduling and software pipelining. It also creases

code in the new loop body, which might increase pressure on register allocation, cause register spilling, and thus cause a loss in performance.

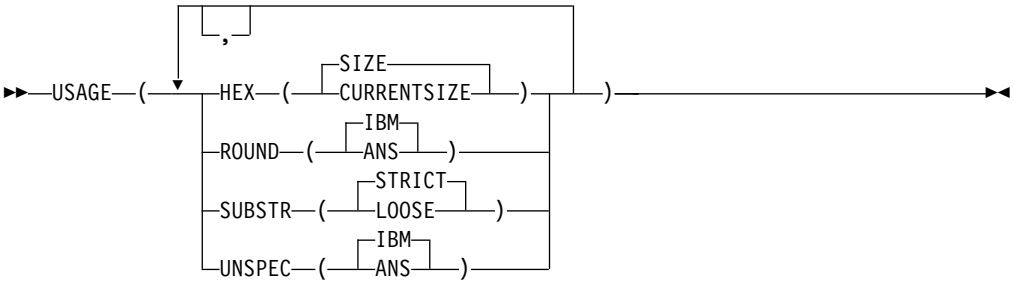
Therefore, before you unroll a loop, take the following steps to check if the UNROLL option improves the performance of a particular application:

1. Compile the program with the usual options.
2. Run the program with a representative workload.
3. Recompile the program with the UNROLL option.
4. Rerun the program under the same conditions.

UNROLL(AUTO) is the default.

USAGE

Using the USAGE option, you can choose different semantics for selected built-in functions.



HEX(SIZE | CURRENTSIZE)
Under the HEX(SIZE) suboption, when HEX is applied to a VARYING or VARYINGZ string, it will a return a hex string that represents the maximum amount of storage used by the string.

Under the HEX(CURRENTSIZE) suboption, when HEX is applied to a VARYING or VARYINGZ string, it will a return a hex string that represents the current amount of storage used by the string.

ROUND(IBM | ANS)
Under the ROUND(IBM) suboption, the second argument to the ROUND built-in function is ignored if the first argument has the FLOAT attribute.

Under the ROUND(ANS) suboption, the ROUND built-in function is implemented as described in the *PL/I Language Reference*.

SUBSTR(STRICT | LOOSE)
Under the SUBSTR(STRICT) suboption, if x has CHARACTER type, a SUBSTR(x,y,z) built-in function reference will return a string whose length is equal to MIN(z, MAXLENGTH(x)).

Under the SUBSTR(LOOSE) suboption, the same reference will return a string whose length is z.

The SUBSTR(LOOSE) suboption might be useful for those who have SUBSTR(x,y,z) references where x is a CHAR(1) BASED variable.

UNSPEC(IBM | ANS)
Under the UNSPEC(IBM) suboption, UNSPEC cannot be applied to a structure, and if applied to an array, returns an array of bit strings.

Under the UNSPEC(ANS) suboption, UNSPEC can be applied to structures, and when applied to a structure or an array, UNSPEC returns a single bit string.

Default: USAGE(HEX(SIZE) ROUND(IBM) SUBSTR(STRICT) UNSPEC(IBM))

WIDECHAR

The WIDECHAR option specifies the format in which WIDECHAR data will be stored.

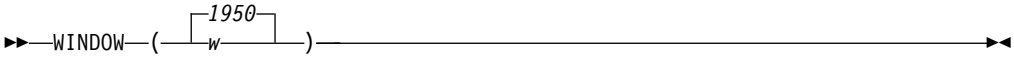
BIGENDIAN
Indicates that WIDECHAR data will be stored in big-endian format. For instance, the WIDECHAR value for the UTF-16 character 1 will be stored as '0031'x.

LITTLEENDIAN
Indicates that WIDECHAR data will be stored in little-endian format. For instance, the WIDECHAR value for the UTF-16 character 1 will be stored as '3100'x.

WX constants should always be specified in big-endian format. Thus the value '1' should always be specified as '0031'wx, even if under the WIDECHAR(LITTLEENDIAN) option, it is stored as '3100'x.

WINDOW

The WINDOW option sets the value for the `w` window argument used in various date-related built-in functions.



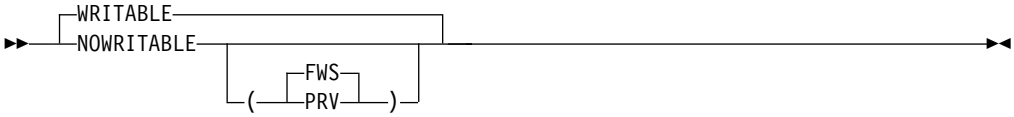
w Either an unsigned integer that represents the start of a fixed window or a negative integer that specifies a "sliding" window. For example, `WINDOW(-20)` indicates a window that starts 20 years before the year when the program runs.

WRITABLE

The WRITABLE option specifies that the compiler can treat static storage as writable (and if it does, this will make the resultant code nonreentrant).

This option has no effect on programs compiled with the RENT option.

Note: The WRITABLE option is ignored under the LP(64) option.



The NORENT WRITABLE options allow the compiler to write on static storage to implement the following constants or variables:

- CONTROLLED variables
- FETCHABLE ENTRY constants
- FILE constants

Under the NORENT WRITABLE options, a module using CONTROLLED variables, performing I/O, or using FETCH is not reentrant.

The NORENT NOWRITABLE options require the compiler not to write on static storage for the following constants or variables:

- CONTROLLED variables
- FETCHABLE ENTRY constants
- FILE constants

Under the NORENT NOWRITABLE options, a module using CONTROLLED variables, performing I/O, or using FETCH is reentrant.

The FWS and PRV suboptions determine how the compiler handles CONTROLLED variables:

FWS
Upon entry to an EXTERNAL procedure, the compiler makes a library call to find storage it can use to address the CONTROLLED variables in that procedure (and any subprocedures).

PRV
The compiler will use the same pseudoregister variable mechanism used by the old OS PL/I compiler to address CONTROLLED variables.

Hence, under the NORENT NOWRITABLE(PRV) options, old and new code can share CONTROLLED variables.

However, this also means that under the NORENT NOWRITABLE(PRV) options, the use of CONTROLLED variables is subject to all the same restrictions as under the old compiler.

Under the NORENT NOWRITABLE(FWS) options, the following application might not perform as well as if they were compiled with the RENT or WRITABLE options:

- Applications that use CONTROLLED variables
- Applications that assign FILE CONSTANTs to FILE VARIABLEs

The performance of an application under NORENT NOWRITABLE(FWS) might be especially bad if it uses many CONTROLLED variables in many PROCEDUREs.

Under the NOWRITABLE option, the following variables and constants cannot be declared in a PACKAGE outside a PROCEDURE:

- CONTROLLED variables
- FETCHABLE ENTRY constants
- FILE constants

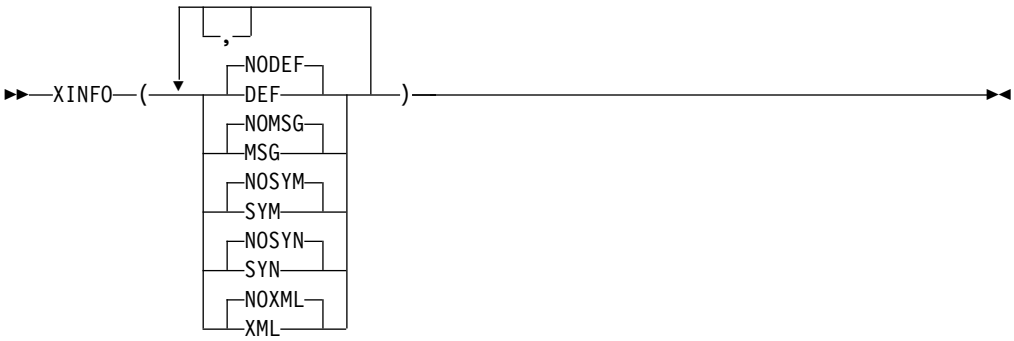
Code compiled with NORENT WRITABLE cannot be mixed with code compiled with NORENT NOWRITABLE if they share any external CONTROLLED variables. In general, you should avoid mixing code compiled with WRITABLE with code compiled with NOWRITABLE.

Related information:

"RENT" on page 68
Your code is "naturally reentrant" if it does not alter any of its static variables. The RENT option specifies that the compiler is to take code that is not naturally reentrant and make it reentrant.

XINFO

The XINFO option specifies that the compiler should generate additional files with extra information about the current compilation unit.



DEF

A definition side-deck file is created. This file lists the following information for the compilation unit:

- All defined EXTERNAL procedures
- All defined EXTERNAL variables
- All statically referenced EXTERNAL routines and variables
- All dynamically called fetched modules

Under batch, this file is written to the file specified by the SYSDEFSD DD statement. Under z/OS UNIX System Services, this file is written to the same directory as the object deck and has the extension `def`.

For instance, given the program:

```
defs: proc;
   dcl (b,c) ext entry;
   dcl x ext fixed bin(31) init(1729);
   dcl y ext fixed bin(31) reserved;
   call b(y);
   fetch c;
   call c;
end;
```

The following def file would be produced:

```
EXPORTS CODE
  DEFS
EXPORTS DATA
  X
IMPORTS
```

```
  B
  Y
FETCH
  C
```

The def file can be used to be build a dependency graph or cross-reference analysis of your application.

NODEF
No definition side-deck file is created.

MSG
Message information is generated to the ADATA file.

Under batch, the ADATA file is generated to the file specified by the SYSADATA DD statement. Under z/OS UNIX, the ADATA is generated in the same directory as the object file and has the extension `adt`.

NOMSG
No message information is generated to the ADATA file. If neither MSG nor SYM is specified, no ADATA file is generated.

SYM
Symbol information is generated to the ADATA file.

Under batch, the ADATA file is generated to the file specified by the SYSADATA DD statement. Under z/OS UNIX, the ADATA file is generated in the same directory as the object file and has the extension `adt`.

NOSYM
No symbol information is generated to the ADATA file.

SYN
Syntax information is generated to the ADATA file. Specifying the XINFO(SYN) option can greatly increase the amount of storage, both in memory and for the file produced, required by the compiler.

Under batch, the ADATA file is generated to the file specified by the SYSADATA DD statement. Under z/OS UNIX, the ADATA file is generated in the same directory as the object file and has the extension `adt`.

NOSYN
No syntax information is generated to the ADATA file.

XML
An XML side-file is created. This XML file includes the following:

- The file reference table for the compilation
- The block structure of the program compiled
- The messages produced during the compilation

Under batch, this file is written to the file specified by the SYSXMLSD DD statement. Under z/OS UNIX System Services, this file is written to the same directory as the object deck and has the extension `xml`.

The DTD file for the XML produced is as follows:

```
<?xml encoding="UTF-8"?>

<!ELEMENT PACKAGE ((PROCEDURE)*,(MESSAGE)*,FILEREFERNCETABLE)>
<!ELEMENT PROCEDURE (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT BEGINBLOCK (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT MESSAGE (MSGNUMBER,MSGLINE?,MSGFILE?,MSGTEXT)>
<!ELEMENT FILE (FILENUMBER,INCLUDEDFROMFILE?,INCLUDEDONLINE?,FILENAME)>
<!ELEMENT FILEREFERENCETABLE (FILECOUNT,FILE+)>
```

```
<!ELEMENT BLOCKFILE (#PCDATA)>
<!ELEMENT BLOCKLINE (#PCDATA)>
<!ELEMENT MSGNUMBER (#PCDATA)>
<!ELEMENT MSGLINE (#PCDATA)>
<!ELEMENT MSGFILE (#PCDATA)>
<!ELEMENT MSGTEXT (#PCDATA)>
<!ELEMENT FILECOUNT (#PCDATA)>
<!ELEMENT FILENUMBER (#PCDATA)>
<!ELEMENT FILENAME (#PCDATA)>
<!ELEMENT INCLUDEFROMFILE (#PCDATA)>
<!ELEMENT INCLUDEDONLINE (#PCDATA)>
```

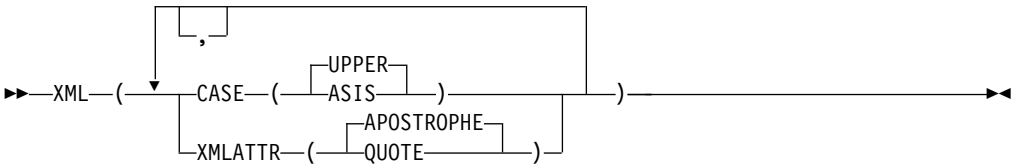
NOXML
No XML side-file is created.

Related information:

"SYSADATA message information," on page 505
When you specify the MSG suboption of the XINFO compile-time option, the compiler generates a SYSADATA file.

XML

The XML option specifies the case of the names in the XML generated by the XMLCHAR built-in function.



CASE(UPPER | ASIS)
Under the CASE(UPPER) suboption, the names in the XML generated by the XMLCHAR built-in function will all be in uppercase.

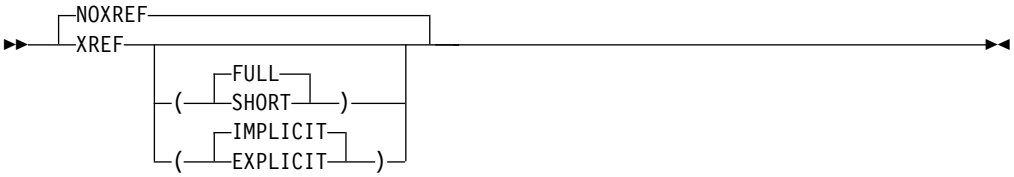
Under the CASE(ASIS) suboption, the names in the XML generated by the XMLCHAR built-in function will be in the case used in their declarations. Note that if you use the MACRO preprocessor without using the macro preprocessor option CASE(ASIS), the source seen by the compiler will have all the names in uppercase - and that would make specifying the XML(CASE(ASIS)) option useless.

XMLATTR(APOSTROPHE | QUOTE)
Under the XMLATTR(APOSTROPHE) suboption, all XML attributes generated by the XMLCHAR built-in function are enclosed in apostrophes.

Under the XMLATTR(QUOTE) suboption, all XML attributes generated by the XMLCHAR built-in function are enclosed in quotation marks. Under CODEPAGE(1026) and COEPAGE(1155), the hex value for the quote character is 'FC'x, and under all other supported EBCDIC code pages, it is '7F'x.

XREF

The XREF option provides a cross-reference table of names used in the program together with the numbers of the statements in which they are declared or referenced in the compiler listing.



ABBREVIATIONS: X, NX

FULL
Under XREF(FULL), all identifiers and attributes are included in the compiler listing. FULL is the default.

SHORT
Under XREF(SHORT), unreferenced identifiers are omitted from the compiler listing.

EXPLICIT
Under XREF(EXPLICIT), a reference to a structure causes only that structure to be included in the compiler listing.

IMPLICIT
Under XREF(IMPLICIT), a reference to a structure causes the structure and all of its members to be included in the compiler listing. IMPLICIT is the default.

The only names not included in the cross reference listing created when you use the XREF option are label references on END statements. For example, assume that statement number 20 in the procedure PROC1 is `END PROC1;`. In this situation, statement number 20 does not appear in the cross reference listing for PROC1.

If you specify both the XREF and ATTRIBUTES options, the two listings are combined. If there is a conflict between SHORT and FULL, the usage is determined by the last option specified. For example, ATTRIBUTES(SHORT) XREF(FULL) results in the FULL option for the combined listing.

Related information:

"Cross-reference table" on page 105
If you specify ATTRIBUTES and XREF, the cross-reference table and the attribute table are combined. The list of attributes for a name is identified by the file number and the line number.

Blanks, comments and strings in options

Wherever you can use a blank when specifying an option, you can also specify as many blanks or comments as you wish. However, there are a few rules that you must take notice of.

If a comment is specified in %PROCESS line or in a line in an options file, the comment must end on the same line as which it begins.

Similarly, if a comment starts in the command line or in the PARM= specification, it must also end there.

The same rule applies to strings: if a string is specified in %PROCESS line or in a line in an options file, the string must end on the same line as which it begins. Similarly, if a string starts in the command line or in the PARM= specification, it must also end there.

Changing the default options

If you want to change the supplied default compiler options, during installation of the compiler, you should edit and submit sample job IBMZWIOP.

This job will let you specify options that will be applied before any other options, thus effectively changing the default options. This job will also let you specify options that will be applied after all other options, thus effectively changing the default options and preventing them from being overridden.

If you want to change the defaults for the macro preprocessor options, you can also do this at installation time by specifying the appropriate PPMACRO option as part of this job. The PPCICS and PPSQL options let you make the corresponding changes for the CICS and SQL preprocessors respectively.

Consult the instructions in the sample job for more information.

Specifying options in the %PROCESS or *PROCESS statements

The %PROCESS or *PROCESS statement identifies the start of each external procedure and allows compiler options to be specified for each compilation. You can use either %PROCESS or *PROCESS in your program; they are equally acceptable.

Note: For consistency and readability, %PROCESS or *PROCESS statements are always referred to as %PROCESS.

The options you specify in adjacent %PROCESS statements apply to the compilation of the source statements to the end of input or the next %PROCESS statement.

To specify options in the %PROCESS statement, code as follows:

```
%PROCESS options;
```

where *options* is a list of compiler options.

You must end the list of options with a semicolon, and the options list should not extend beyond the default right-hand source margin. The percent sign (%) or asterisk (*) must appear in the first column of the record. The keyword PROCESS can follow in the next byte (column) or after any number of blanks. You must separate option keywords by a comma or at least one blank.

The number of characters is limited only by the length of the record. If you do not wish to specify any options, code as follows:

```
%PROCESS;
```

If you find it necessary to continue the %PROCESS statement onto the next record, terminate the first part of the list after any delimiter, and continue on the next record. You cannot split keywords or keyword arguments across records. You can continue a %PROCESS statement on several lines, or start a new %PROCESS statement. The following example shows multiple adjacent %PROCESS statements:

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST TEST ;
```

Compile-time options, their abbreviated syntax, and their IBM-supplied defaults are shown in Table 3 on page 4.

How the compiler determines whether there are any %PROCESS statements depends on the format of the initial source file:

F or FB format
If the first character in the record is an asterisk (*) or a percent sign (%), the compiler will check whether the next nonblank characters are PROCESS.

V or VB format
If the first character in the record is a numeric, the compiler will assume that the first 8 characters are sequence numbers, and if the ninth character is an asterisk (*) or a percent sign (%), it will check whether the next nonblank characters are PROCESS. However, if the first character is not a numeric but an asterisk (*) or a percent sign (%), the compiler will check if the next nonblank characters are PROCESS.

U format
If the first character in the record is an asterisk (*) or a percent sign (%), the compiler will check if the next nonblank characters are PROCESS.

Using % statements

Statements that direct the operation of the compiler begin with a percent (%) symbol. You can use % statements to control the source program listing and to include external strings in the source program. % statements must not have label or condition prefixes and cannot be a unit of a compound statement. You should place each % statement on a line by itself.

The usage of each % control statement is listed below. For a complete description of these statements, see the *PL/I Language Reference*.

%INCLUDE
Directs the compiler to incorporate external text into the source program.

%XINCLUDE
Directs the compiler to incorporate external text into the source program if it has not been previously included.

%PRINT
Directs the compiler to resume printing the source and insource listings.

%NOPRINT
Directs the compiler to suspend printing the source and insource listings until a %PRINT statement is encountered.

%PAGE
Directs the compiler to print the statement immediately after a %PAGE statement in the program listing on the first line of the next page.

%POP Directs the compiler to restore the status of the %PRINT and %NOPRINT saved by the most recent %PUSH.

%PUSH
Saves the current status of the %PRINT and %NOPRINT in a *push down* stack on a last-in, first-out basis.

%SKIP
Specifies the number of lines to be skipped.

Using the %INCLUDE statement

%INCLUDE statements are used to include additional PL/I files at specified points in a compilation unit.

For information about how to use the %INCLUDE statement to incorporate source text from a library into a PL/I program, see the *PL/I Language Reference*.

For a batch compilation

A *library* is an z/OS partitioned data set that can be used to store other data sets called members. Source text that you might want to insert into a PL/I program using a %INCLUDE statement must exist as a member within a library. For further information about the process of defining a source statement library to the compiler, see "Source Statement Library (SYSLIB)" on page 164.

The statement `%INCLUDE DD1 (INVERT);` specifies that the source statements in member INVERT of the library defined by the DD statement with the name DD1 are to be inserted consecutively into the source program. The compilation job step must include appropriate DD statements.

If you omit the ddname, the ddname SYSLIB is assumed. In such a case, you must include a DD statement with the name SYSLIB. (The IBM-supplied cataloged procedures do not include a DD statement with this name in the compilation procedure step.)

For a z/OS UNIX compilation

The name of the actual include file must be lowercase, unless you specify UPPERINC. For example, if you use the include statement `%include sample`, the compiler can find the file `sample.inc` but cannot find the file `SAMPLE.inc`. Even if you use the include statement `%include SAMPLE`, the compiler still looks for `sample.inc`.

The compiler looks for INCLUDE files in the following order:

1. Current directory
2. Directories specified with the –I flag or with the INCDIR compiler option
3. `/usr/include` directory
4. PDS specified with the INCPDS compiler option

The first file found by the compiler is included into your source.

A %PROCESS statement in source text included by a %INCLUDE statement results in an error in the compilation.

Figure 1 on page 102 shows the use of a %INCLUDE statement to include the source statements for FUN in the procedure TEST. The library HPU8.NEWLIB is defined in the DD statement with the qualified name PLI.SYSLIB, which is added to the statements of the cataloged procedure for this job. Because the source statement library is defined by a DD statement with the name SYSLIB, the %INCLUDE statement need not include a ddname.

It is not necessary to invoke the preprocessor if your source program and any text to be included do not contain any macro statements.

```
//OPT4#9     JOB
//STEP3      EXEC IBMZCBG,PARM.PLI='INC,S,A,X,NEST'
//PLI.SYSLIB DD DSN=HPU8.NEWLIB,DISP=OLD
//PLI.SYSIN  DD *
   TEST: PROC OPTIONS(MAIN) REORDER;
     DCL ZIP PIC '99999';           /* ZIP CODE                      */
     DCL EOF BIT INIT('0'B);
     ON ENDFILE(SYSIN) EOF = '1'B;
     GET EDIT(ZIP) (COL(1), P'99999');
     DO WHILE(¬EOF);
       PUT SKIP EDIT(ZIP, CITYFUN(ZIP)) (P'99999', A(16));
       GET EDIT(ZIP) (COL(1), P'99999');
     END;
     %PAGE;
     %INCLUDE FUN;
   END;                             /* TEST                          */
//GO.SYSIN DD *
95141
95030
94101
//
```

Figure 1. Including source statements from a library

Using the compiler listing

During compilation, the compiler generates a listing, most of which is optional, that contains information about the source program, the compilation, and the object module.

The following description of the listing refers to its appearance on a printed page.

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

If compilation terminates before reaching a particular stage of processing, the corresponding listings do not appear.

Heading information

The first page of the listing is identified by the product number, the compiler version number, a string specifying when the compiler was built, and the date and the time compilation began. This page and subsequent pages are numbered.

The listing will then show any options that have been specified for this compilation. These options will be shown even if the NOOPTIONS option has been specified and will include, in order, the following options:

- The initial install options (those are the options set at install time and which are applied before any other options)
- Under z/OS UNIX, any options specified in the IBM_OPTIONS environment variable
- Any options specified in the parameter string passed to the compiler (that is, in the command line under z/OS UNIX or in the PARM= under batch)
- Any options specified in options files named in the compiler parameter string

This will include the name of each options file and its contents in the same form as read by the compiler.
- Any options specified in *PROCESS or %PROCESS lines in the source
- The final install options (those are the options set at installation and which are applied after any other options)

Near the end of the listing you will find either a statement that no errors or warning conditions were detected during the compilation, or a message that one or more errors were detected. For information about the format of the messages, see "Messages and return codes" on page 110. The second to the last line of the listing shows the time taken for the compilation. The last line of the listing is `END OF COMPILATION OF` *xxxx*, where *xxxx* is the external procedure name. If you specify the NOSYNTAX compiler option, or if the compiler aborts early in the compilation, the external procedure name *xxxx* is not included and the line truncates to `END OF COMPILATION`.

The following topics describe the optional parts of the listing in the order in which they appear.

Options used for compilation

If you specify the OPTIONS option, a complete list of the options specified for the compilation, including the default options, appears on the following pages.

The listing shows the settings of all the options finally in effect during the compilation. If the setting of an option differs from the default setting after the initial install options were applied, that line is marked with a plus sign (+).

Preprocessor input

If you specify both the MACRO and INSOURCE options, the compiler lists input to the preprocessor, one record per line, each line numbered sequentially at the left.

If the preprocessor detects an error or the possibility of an error, it prints a message on the page or pages following the input listing. The format of these messages is the same as the format for the compiler messages described in "Messages and return codes" on page 110.

SOURCE program

If you specify the SOURCE option, the compiler lists one record per line. These records always include the source line and file numbers. However, if a file contains 999999 or more lines, the compiler flags the file as too large and lists only the last 6 digits in the source line numbers for that file.

If the input records contain printer control characters, or %SKIP or %PAGE statements, the lines are spaced accordingly.

Use the %NOPRINT statement to stop the printing of the listing.

Use the %PRINT statement to restart the printing of the listing.

If you specify the MACRO option, the source listing shows the included text in place of the %INCLUDE statements in the primary input data set.

Statement nesting level

If you specify the NEST option, the block level and the DO-level are printed to the right of the statement or line number under the headings LEV and NT respectively.

See the following example:

```
Line.File LV NT
   1.0             A: PROC OPTIONS(MAIN);
   2.0     1       B: PROC;
   3.0     2          DCL K(10,10) FIXED BIN (15);
   4.0     2          DCL Y FIXED BIN (15) INIT (6);
   5.0     2          DO I=1 TO 10;
   6.0     2  1          DO J=1 TO 10;
   7.0     2  2             K(I,J) = N;
   8.0     2  2          END;
   9.0     2  1       BEGIN;
  10.0     3  1          K(1,1)=Y;
  11.0     3  1       END;
  12.0     2  1    END B;
  13.0     1       END A;
```

ATTRIBUTE and cross-reference table

If you specify the ATTRIBUTES option, the compiler prints an attribute table containing a list of the identifiers in the source program together with their declared and default attributes. If you specify the XREF option, the compiler prints a cross-reference table containing a list of the identifiers in the source program together with the file and line numbers of the statements in which they appear.

If you specify both ATTRIBUTES and XREF, the two tables are combined. In these tables, if you explicitly declare an identifier, the compiler will list file number and line number of its DECLARE. Contextually declared variables are marked by +++++, and other implicitly declared variables are marked by *****.

Attribute table

The attribute table contains a list of the identifiers in the source program together with their declared and default attributes.

The compiler never includes the attributes INTERNAL and REAL. You can assume them unless the respective conflicting attributes, EXTERNAL and COMPLEX, appear.

For a file identifier, the attribute FILE always appears, and the attribute EXTERNAL appears if it applies; otherwise, the compiler lists only explicitly declared attributes.

The OPTIONS attribute will not appear unless the ENTRY attribute applies, and then only the following options will appear as appropriate:

- ASSEMBLER
- COBOL
- FETCHABLE
- FORTRAN
- NODESCRIPTOR
- RETCODE

The compiler prints the dimension attribute for an array first. It prints the bounds as in the array declaration, but expressions are replaced by asterisks unless they have been reduced by the compiler to a constant, in which case the value of the constant is shown.

For a character string, a bit string, a graphic string, or an area variable, the compiler prints the length, as in the declaration, but expressions are replaced by asterisks unless they have been reduced by the compiler to a constant, in which case the value of the constant is shown.

Cross-reference table

If you specify ATTRIBUTES and XREF, the cross-reference table and the attribute table are combined. The list of attributes for a name is identified by the file number and the line number.

An identifier appears in the `Sets:` part of the cross-reference table under the following conditions:

- It is the target of an assignment statement.
- It is used as a loop control variable in DO loops.
- It is used in the SET option of an ALLOCATE or LOCATE statement.
- It is used in the REPLY option of a DISPLAY statement.

If there are unreferenced identifiers, they are displayed in a separate table.

Aggregate length table

An aggregate length table is obtained by the AGGREGATE option. The table includes structures but not arrays that have non-constant extents, but the sizes and offsets of elements within structures with non-constant extents can be inaccurate or specified as *.

For the aggregates listed, the table contains the following information:

- Where the aggregate is declared
- The name of the aggregate and each element within the aggregate
- The byte offset of each element from the beginning of the aggregate
- The length of each element
- The total length of each aggregate, structure, and substructure
- The total number of dimensions for each element

Be careful when interpreting the data offsets indicated in the data length table. An odd offset does not necessarily represent a data element without halfword, fullword, or even double word alignment. If you specify or infer the aligned attribute for a structure or its elements, the proper alignment requirements are consistent with respect to other elements in the structure, even though the table does not indicate the proper alignment relative to the beginning of the table.

If there is padding between two structure elements, a /*PADDING*/ comment appears with appropriate diagnostic information.

Statement offset addresses

If the LIST compile option is used, the compiler includes a pseudo-assembler listing in the compiler listing. This listing includes, for each instruction, an offset whose meaning depends on the setting of the BLKOFF compiler option.

- Under the BLKOFF option, this offset is the offset of the instruction from the primary entry point for the function or subroutine to which it belongs. Thus under this option, the offsets are reset with each new block.
- Under the NOBLKOFF option, this offset is the offset of the instruction from the start of the compilation unit. Thus under this option, the offsets are cumulative.

The pseudo-assembler listing also includes, at the end of the code for each block, the offset of the block from the start of the current module (so that the offsets shown for each statement can be translated to either block or module offsets).

These offsets can be used with the offset given in a runtime error message to determine the statement to which the message applies.

The OFFSET option produces a table that gives for each statement, the offset of the first instruction belonging to that statement.

In the example shown in Figure 2 on page 107, the message indicates that the condition was raised at offset +98 from the SUB1 entry. The compiler listing excerpt shows this offset associated with line number 8. The runtime output from this erroneous statement is shown if Figure 3 on page 107.

```
Compiler Source
    Line.File
       1.0
       2.0      TheMain: proc options( main );
       3.0        call sub1();
       4.0        Sub1: proc;
       5.0          dcl (i, j) fixed bin(31);
       6.0
       7.0          i = 0; j = 0;
       8.0          j = j / i;
       9.0          put skip data( j );
      10.0        end Sub1;
      11.0      end TheMain;

 . . .

OFFSET OBJECT CODE       LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

000000                   000002 |        THEMAIN  DS       0D
000000  47F0  F024       000002 |                 B        36(,r15)
000004  01C3C5C5                                           CEE eyecatcher
000008  000000B0                                           DSA size
00000C  000001F8                                           =A(PPA1-THEMAIN)
000010  47F0  F001       000002 |                 B        1(,r15)

 . . .

000000                   000004 |        SUB1     DS       0D
000000  47F0  F024       000004 |                 B        36(,r15)
000004  01C3C5C5                                           CEE eyecatcher
000008  00000140                                           DSA size
00000C  00000190                                           =A(PPA1-SUB1)
000010  47F0  F001       000004 |                 B        1(,r15)

...

000086  5020  D0B8       000007 |                 ST       r2,I(,r13,184)
00008A  1842             000007 |                 LR       r4,r2
00008C  5040  D0BC       000007 |                 ST       r4,J(,r13,188)
000090  5800  D0B8       000008 |                 L        r0,I(,r13,184)
000094  8E40  0020       000008 |                 SRDA     r4,32
000098  1D40             000008 |                 DR       r4,r0
00009A  1805             000008 |                 LR       r0,r5
00009C  5000  D0BC       000008 |                 ST       r0,J(,r13,188)
0000A0  4100  D0C0       000009 |                 LA       r0,_temp1(,r13,192)
0000A4  5000  D130       000009 |                 ST       r0,_temp2(,r13,304)
0000A8  A708  5A88       000009 |                 LHI      r0,H'23176'
0000AC  4000  D0EC       000009 |                 STH      r0,_temp1(,r13,236)
0000B0  5800  6004       000009 |                 L        r0,SYSPRINT(,r6,4)
0000B4  5000  D12C       000009 |                 ST       r0,_temp2(,r13,300)
0000B8  4100  0001       000009 |                 LA       r0,1
0000BC  5000  D0C0       000009 |                 ST       r0,_temp1(,r13,192)
0000C0  4100  D128       000009 |                 LA       r0,_temp2(,r13,296)

...
```

Figure 2. Finding statement number (compiler listing example)

```
Message :

IBM0301S ONCODE=320  The ZERODIVIDE condition was raised.
         From entry point SUB1 at compile unit offset +00000098
         at entry offset +00000098 at address 0EB00938.
```

Figure 3. Finding statement number (runtime message example)

Entry offsets given in dump and ON-unit SNAP error messages can be compared with this table and the erroneous statement discovered. The statement is identified by finding the section of the table that relates to the block named in the message and then finding the largest offset less than or equal to the offset in the message. The statement number associated with this offset is the one needed.

Storage offset listing

If the MAP compile option is used, the compiler includes a storage offset listing in the compiler listing.

This listing gives the location in storage of the following level-1 variables if they are used in the program:

- AUTOMATIC
- CONTROLLED except for PARAMETERs
- STATIC except for ENTRY CONSTANTs that are not FETCHABLE

The listing might also include some compiler generated temporaries.

For an AUTOMATIC variable with adjustable extents, there will be two entries in this table:

- An entry with `_addr` prefixing the variable name - this entry gives the location of the address of the variable
- An entry with `_desc` prefixing the variable name - this entry gives the location of the address of the variable's descriptor

For STATIC and CONTROLLED variables, the storage location will depend on the RENT/NORENT compiler option, and if the NORENT option is in effect, the location of CONTROLLED variables will also depend on the WRITABLE/ NOWRITABLE compiler option.

The first column in the Storage Offset Listing is labeled IDENTIFIER and holds the name of the variable whose location appears in the fourth column.

The second column in the Storage Offset Listing is labeled DEFINITION and holds a string in the format *"B-F:N"*, where

- *B* is the number of the block where the variable is declared.

  You can find the name of the block corresponding to this block number in the Block Name List, which will proceed the Storage Offset Listing (and the Pseudo Assembly Listing, if any).
- *F* is the number of the source file where the variable is declared.

  You can find the name of the file corresponding to this file number in the File Reference Table, which will appear very near the end of the entire compilation listing.
- *N* is the number of the source line where the variable is declared in that source file.

The third column in the Storage Offset Listing is labeled ATTRIBUTES and indicates the storage class of the variable.

The fourth column in the Storage Offset Listing is unlabeled and tells how to find the location of the variable.

This storage offset listing is sorted by block and by variable name, and it also includes only user variables. However, specifying the MAP option also causes the compiler to produce the following maps:

- A "static map" that lists all STATIC variables but sorted by hex offset
- An "automatic map" that lists, for each block, all AUTOMATIC variables but sorted by hex offset

The mapping rules of the PL/I language might require that a structure be offset by up to 8 bytes from where it would seem to start. For example, consider the AUTOMATIC structure A declared as follows:

```
dcl
  1 A,
    2 B char(2),
    2 C fixed bin(31);
```

Because C must be aligned on a 4-byte boundary, 2 bytes of padding will be needed for this structure. However, PL/I places those 2 bytes not after B, but before B. These 2 bytes of "padding" before a structure starts are referred to as the *hang bytes* for the structure.

These hang bytes will also be reflected in the "automatic map" generated by the compiler. The "storage offset listing" will show the offset and length for A without including its hang bytes:

```
A     Class = automatic,   Location = 186 : 0xBA(r13),     Length = 6
```

In contrast, the "automatic map" will show the offset and length for A with its hang bytes included:

```
OFFSET (HEX)   LENGTH (HEX)   NAME

     98              8        #MX_TEMP1
     A0             18        _Sfi
     B8              8        A
```

Expressions and attributes listing

If you use the DECOMP compiler option, the compiler includes, in the compiler listing, a section that shows all intermediate expressions and their attributes for all expressions used in the source program.

Related information:

"DECOMP" on page 22
The DECOMP option instructs the compiler to generate a listing section that gives a decomposition of expressions used in the compilation.

File reference table

The file reference table provides information about the files read during the compilation.

The file reference table consists of three columns that list the following file information:

- The number assigned by the compiler to the file
- The included-from data for the file
- The name of the file

The first entry in the included-from column is blank because the first file listed is the source file. Subsequent entries in this column show the line number of the include statement followed by a period and the file number of the source file containing the include statement.

If the file is a member of a PDS or PDSE, the file name lists the fully qualified dataset name and the member name.

If the file is included by a subsystem (such as Librarian), the file name will have the form `DD:ddname(member)`, where

- *ddname* is the ddname specified on the %INCLUDE statement (or SYSLIB if no ddname was specified).
- *member* is the member name specified on the %INCLUDE statement.

Messages and return codes

If the preprocessor or the compiler detects an error, or the possibility of an error, messages are generated. For every compilation job or job step, the compiler generates a return code that indicates the degree of success or failure.

Messages

Messages generated by the preprocessor appear in the listing immediately after the listing of the statements processed by the preprocessor. You can generate your own messages in the preprocessing stage by using the %NOTE statement. Such messages might be used to show how many times a particular replacement had been made. Messages generated by the compiler appear at the end of the listing.

For compilations that produce no messages, the compiler will include a line saying `no compiler messages` where the compiler messages would have been listed.

Messages are displayed in the following format:

```
PPPnnnnI X
```

PPP
Is the prefix identifying the origin of the message (for example, `IBM` indicates the PL/I compiler).

nnnn
Is the 4-digit message number.

X Identifies the severity code.

All messages are graded according to their severity, and the severity codes are I, W, E, S, and U.

The compiler lists only messages that have a severity equal to or greater than that specified by the FLAG option, as shown in Table 6.

Table 6. Using the FLAG option to select the lowest message severity listed

Type of message	Option
Information	FLAG(I)
Warning	FLAG(W)
Error	FLAG(E)
Severe Error	FLAG(S)
Unrecoverable Error	Always listed

For information about the text of each message, an explanation of each message, and any recommended programmer response for the message, see the *Enterprise PL/I Messages and Codes*.

Return codes

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved. For z/OS, this code appears in the *end-of-step* message that follows the listing of the job control statements and job scheduler messages for each step.

Table 7 provides an explanation of each severity code and its comparable return code.

Table 7. Description of PL/I error codes and return codes

Severity code	Return code	Message type	Description
I	0000	Informational	The compiled program should run correctly. The compiler might inform you of a possible inefficiency in your code or some other condition of interest.
W	0004	Warning	A statement might be in error (warning) even though it is syntactically valid. The compiled program should run correctly, but it might produce different results than expected or be significantly inefficient.
E	0008	Error	A simple error fixed by the compiler. The compiled program should run correctly, but it might produce different results than expected.
S	0012	Severe	An error not fixed by the compiler. If the program is compiled and an object module is produced, it should not be used.
U	0016	Unrecoverable	An error that forces termination of the compilation. An object module is not successfully created.

Note: Compiler messages are printed in groups according to these severity levels.

Example

This example of the compiler listing is generated when the compiler compiles the following `msgsumm` program with these options: PP(SQL,MACRO,CICS), SOURCE, FLAG(I), INSOURCE, MSGSUMMARY(XREF).

```
msgsumm: proc;

  exec sql include sqlca;

  exec cics what now;

  exec cics not this;

  %dcl z0 fixed bin;
  %dcl z1 fixed dec;
end;
```

Note: The program is intentionally incorrect. Because the MSGSUMMARY option is specified, the compiler includes the `Summary of Messages` section at the end of the listing. This section also includes the line numbers associated with each of the messages in the summary because the XREF suboption to the MSGSUMMARY option is specified.

```
5655-PL5  IBM(R) Enterprise PL/I for z/OS       V5.R1.M0 (Built:20160513)
                                                2016.05.13 14:19:23 Page 1
                     Options Specified
  Install:
  Command: +DD:OPTIONS
  File: DD:OPTIONS
    PP(SQL,MACRO,CICS),S,F(I),IS,MSGSUMMARY(XREF)
  Install:

5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 2
 SQL (Built:20160421) Preprocessor Source
    Line.File
       1.0
       2.0      msgsumm: proc;
       3.0
       4.0        exec sql include sqlca;
       5.0
       6.0        exec cics what now;
       7.0        exec cics not this;
       8.0
       9.0        %dcl z0 fixed bin;
      10.0        %dcl z1 fixed dec;
      11.0      end;

5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 3
 SQL Preprocessor Options Used
     CCSID0
   NOCODEPAGE
     DEPRECATE( STMT() )
   NOEMPTYDBRM
   NOINCONLY
   NOWARNDECP
 DB2 for z/OS Coprocessor Options Used
   APOST
   APOSTSQL
   ATTACH(TSO)
   CCSID(500)
   CONNECT(2)
   DEC(15)
   FLOAT(S390)
   NEWFUN(YES)
   TWOPASS
   PERIOD
   STDSQL(NO)
   SQL(DB2)
   NO XREF
   NO SOURCE
   DSNHDECP LOADED FROM - (DSN910.SDSNLOAD(DSNHDECP)

5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 4
 SQL Preprocessor Messages
 Message       Line.File Message Description
```

Figure 4. Compiler listing example

```
5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 5
 MACRO (Built:20160111) Preprocessor Source
    Line.File
       1.0
       2.0      msgsumm: proc;
       2.0
       3.0
       4.0
       4.0      /*$*$*$
       4.0        exec sql include sqlca
       4.0       $*$*$*/
       4.0      DCL
       4.0        1 SQLCA  ,
       4.0          2 SQLCAID     CHAR(8),
       4.0          2 SQLCABC     FIXED BIN(31),
       4.0          2 SQLCODE     FIXED BIN(31),
       4.0          2 SQLERRM     CHAR(70) VAR,
       4.0          2 SQLERRP     CHAR(8),
       4.0          2 SQLERRD(6)  FIXED BIN(31),
       4.0          2 SQLWARN,
       4.0            3 SQLWARN0  CHAR(1),
       4.0            3 SQLWARN1  CHAR(1),
       4.0            3 SQLWARN2  CHAR(1),
       4.0            3 SQLWARN3  CHAR(1),
       4.0            3 SQLWARN4  CHAR(1),
       4.0            3 SQLWARN5  CHAR(1),
       4.0            3 SQLWARN6  CHAR(1),
       4.0            3 SQLWARN7  CHAR(1),
       4.0          2 SQLEXT,
       4.0            3 SQLWARN8  CHAR(1),
       4.0            3 SQLWARN9  CHAR(1),
       4.0            3 SQLWARNA  CHAR(1),
       4.0            3 SQLSTATE  CHAR(5);
       4.0
       5.0
       6.0        exec cics what now;
       7.0        exec cics not this;
       8.0
       9.0        %dcl z0 fixed bin;
      10.0        %dcl z1 fixed dec;
      11.0      end;
```

Compiler listing example (continued)

```
5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 6
 MACRO Messages
 Message       Line.File Message Description
 IBM3552I E       9.0    The statement element BIN is invalid. The statement
                         will be ignored.
 IBM3552I E      10.0    The statement element DEC is invalid. The statement
                         will be ignored.
 IBM3258I W       9.0    Missing ; assumed before BIN.
 IBM3258I W      10.0    Missing ; assumed before DEC.

5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 7
 CICS (Built:20160101) Preprocessor Source
    Line.File
       1.0
       2.0      MSGSUMM: PROC;
       2.0
       3.0
       4.0
       4.0      /*$*$*$
       4.0        exec sql include sqlca
       4.0       $*$*$*/
       4.0      DCL
       4.0        1 SQLCA  ,
       4.0          2 SQLCAID     CHAR(8),
       4.0          2 SQLCABC     FIXED BIN(31),
       4.0          2 SQLCODE     FIXED BIN(31),
       4.0          2 SQLERRM     CHAR(70) VAR,
       4.0          2 SQLERRP     CHAR(8),
       4.0          2 SQLERRD(6)  FIXED BIN(31),
       4.0          2 SQLWARN,
       4.0            3 SQLWARN0  CHAR(1),
       4.0            3 SQLWARN1  CHAR(1),
       4.0            3 SQLWARN2  CHAR(1),
       4.0            3 SQLWARN3  CHAR(1),
       4.0            3 SQLWARN4  CHAR(1),
       4.0            3 SQLWARN5  CHAR(1),
       4.0            3 SQLWARN6  CHAR(1),
       4.0            3 SQLWARN7  CHAR(1),
       4.0          2 SQLEXT,
       4.0            3 SQLWARN8  CHAR(1),
       4.0            3 SQLWARN9  CHAR(1),
       4.0            3 SQLWARNA  CHAR(1),
       4.0            3 SQLSTATE  CHAR(5);
       4.0
       5.0
       6.0        EXEC CICS WHAT NOW;
       7.0        EXEC CICS NOT THIS;
       8.0
      11.0      END;
```

Compiler listing example (continued)

```
5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 8
 CICS Messages
 Message       Line.File Message Description
 IBM3750I S       6.0    DFH7059I S  WHAT COMMAND IS NOT VALID AND IS NOT
                         TRANSLATED.
 IBM3750I S       7.0    DFH7059I S  NOT COMMAND IS NOT VALID AND IS NOT
                         TRANSLATED.

5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 9
 No Compiler Messages
 File Reference Table
   File    Included From  Name
      0                   DD:SYSIN

5655-PL5  IBM(R) Enterprise PL/I for z/OS       2016.05.13 14:19:23 Page 10
 Summary of Messages
 Component Message     Total  Default Message Description
 SQL       IBM3250I W      1  DSNH053I DSNHPSRV  NO SQL STATEMENTS WERE FOUND
                              Refs: 4.0
 MACRO     IBM3552I E      2  The statement element %1 is invalid. The statement
                              will be ignored.
                              Refs: 9.0 10.0
 MACRO     IBM3258I W      2  Missing %1 assumed before %2.
                              Refs: 9.0 10.0
 CICS      IBM3750I S      2  DFH7059I S  WHAT COMMAND IS NOT VALID AND IS NOT
                              TRANSLATED.
                              Refs: 6.0 7.0
 Compiler  <none>

 Component    Return Code    Messages (Total/Suppressed)    Time
 SQL              4               1  /  0                   0 secs
 MACRO            8               4  /  0                   0 secs
 CICS            12               2  /  0                   0 secs
 Compiler         0               0  /  0                   0 secs
 End of compilation
```

Compiler listing example (continued)

Chapter 2. PL/I preprocessors

When you use the PL/I compiler, you can specify one or more of the integrated preprocessors in your program. You can specify the include preprocessor, the macro preprocessor, the SQL preprocessor, or the CICS preprocessor, and specify the order in which you want them to be called.

- The include preprocessor processes special include directives and incorporates external source files.
- The macro preprocessor, based on %statements and macros, modifies your source program.
- The SQL preprocessor modifies your source program and translates EXEC SQL statements into PL/I statements.
- The CICS preprocessor modifies your source program and translates EXEC CICS statements into PL/I statements.

Each preprocessor supports a number of options that you can use to tailor the processing to your needs.

You can set the default options for each of the preprocessors by using the corresponding attributes in the configuration file.

The compile-time options MDECK, INSOURCE, and SYNTAX are meaningful only when you also specify the PP option.

Related information:

"MDECK" on page 54
The MDECK option specifies that the preprocessor produces a copy of its output either on the file defined by the SYSPUNCH DD statement under z/OS or on the .dek file under z/OS UNIX.

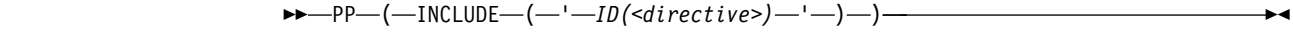
"INSOURCE" on page 43
The INSOURCE option specifies that the compiler should include a listing of the source program before the PL/I macro, CICS, or SQL preprocessors translate it.

"SYNTAX" on page 86
The SYNTAX option specifies that the compiler continues into syntax checking after preprocessing when you specify the MACRO option, unless an unrecoverable error has occurred. Whether the compiler continues with the compilation depends on the severity of the error, as specified by the NOSYNTAX option.

Include preprocessor

The include preprocessor allows you to incorporate external source files into your programs by using include directives other than the PL/I directive %INCLUDE.

The following syntax diagram illustrates the options supported by the INCLUDE preprocessor:



ID Specifies the name of the include directive. Any line that starts with this directive as the first set of nonblank characters is treated as an include directive.

© Copyright IBM Corp. 1999, 2017

The specified directive must be followed by one or more blanks, an include member name, and finally an optional semicolon. Syntax for ddname(membername) is not supported.

In the following example, the first include directive is valid and the second one is not:

```
++include payroll
++include syslib(payroll)
```

Example 1

The following example causes all lines that start with -INC (and possibly preceding blanks) to be treated as include directives:

```
pp( include( 'id(-inc)'))
```

Example 2

The following example causes all lines that start with ++INCLUDE (and possibly preceding blanks) to be treated as include directives:

```
pp( include( 'id(++include)'))
```

Include preprocessor options in the configuration file

You can set the default options for the include preprocessor by using the **pliopt** or **incopt** attribute in the configuration file.

Include preprocessor options environment variable

You can set the default options for the include preprocessor by using the IBM.PPINCLUDE environment variable.

Macro preprocessor

Macros allow you to write commonly used PL/I code in a way that hides implementation details and the data that is manipulated and exposes only the operations. In contrast with a generalized subroutine, macros allow generation of only the code that is needed for each individual use. You can invoke the macro preprocessor by specifying either the MACRO option or the PP(MACRO) option.

You can specify `PP(MACRO)` without any options or with options described in "Macro preprocessor options" on page 119.

The defaults for all these options cause the macro preprocessor to behave the same as the OS PL/I V2R3 macro preprocessor.

If options are specified, the list must be enclosed in quotation marks (single or double, as long as they match); for example, to specify the FIXED(BINARY) option, you must specify PP(MACRO('FIXED(BINARY)')).

If you want to specify more than one option, you must separate them with a comma or one or more blanks. For example, to specify the CASE(ASIS) and RESCAN(UPPER) options, you can specify PP(MACRO('CASE(ASIS) RESCAN(UPPER)')) or PP(MACRO("CASE(ASIS),RESCAN(UPPER)")). You can specify the options in any order.

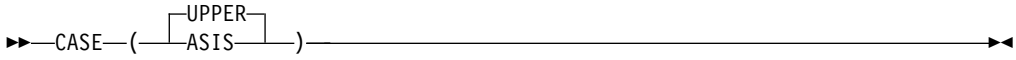
The macro preprocessing facilities of the compiler are described in the *PL/I Language Reference*.

Macro preprocessor options

This section describes the options that the macro preprocessor supports.

CASE

This option specifies whether the preprocessor should convert the input text to uppercase.



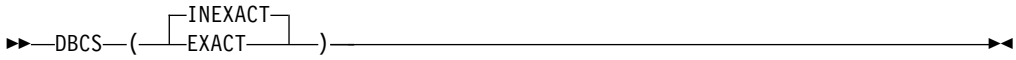
UPPER
The input text is to be converted to uppercase.

ASIS
The input text is left "as is".

Under the GRAPHIC option, CASE(ASIS) is always in effect.

DBCS

This option specifies whether the preprocessor should normalize DBCS during text replacement.

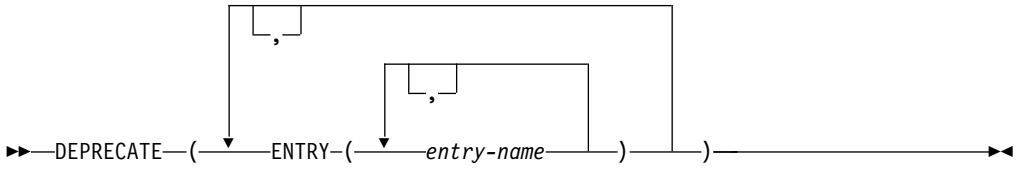


EXACT
The input text is left "as is", and the preprocessor will treat <kk.B> and <kk>B as different names.

INEXACT
The input text is "normalized", and the preprocessor will treat <kk.B> and <kk>B as two versions of the same name.

DEPRECATE

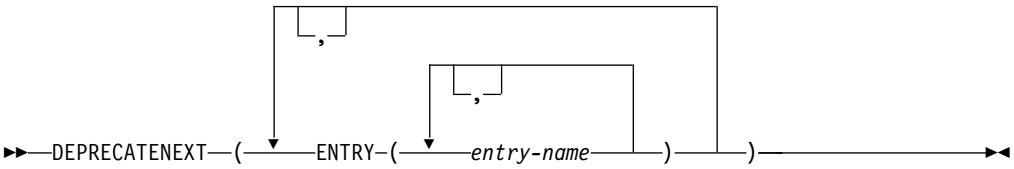
This option flags the usage of macro procedures that you want to deprecate with error messages.



ENTRY
Flags any usage of a macro procedure with name *entry-name*.

DEPRECATENEXT

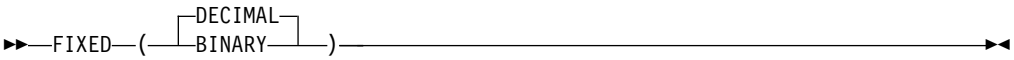
This option flags the usage of macro procedures that you want to deprecate with warning messages.



ENTRY
Flags any usage of a macro procedure with name *entry-name*.

FIXED

This option specifies the default base for FIXED variables.



DECIMAL
FIXED variables will have the attributes REAL FIXED DEC(5).

BINARY
FIXED variables will have the attributes REAL SIGNED FIXED BIN(31).

INCONLY

The INCONLY option specifies that the preprocessor should process only %INCLUDE and %XINCLUDE statements.

The NOINCONLY option specifies that the preprocessor should process all preprocessor statements, not only %INCLUDE and %XINCLUDE statements.

When the INCONLY option is in effect, you can use neither INCLUDE nor XINCLUDE as a macro:

- Procedure name
- Statement label
- Variable name

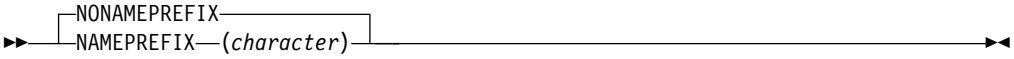
The INCONLY option and the NOINCONLY option are mutually exclusive.

For compatibility, the default is NOINCONLY.

NAMEPREFIX

The NAMEPREFIX option specifies that the names of preprocessor procedures and variables must start with the specified character.

The NONAMEPREFIX option specifies that the names of preprocessor procedures and variables are not required to start with one particular character.



The character should be specified "as is" and should not be enclosed in quotation marks.

The default is NONAMEPREFIX.

RESCAN

This option specifies how the preprocessor should handle the case of identifiers when rescanning text.

UPPER
Rescans will not be case-sensitive.

ASIS
Rescans will be case-sensitive.

To see the effect of this option, consider the following code fragment:

```
%dcl eins char ext;
%dcl text char ext;

%eins = 'zwei';

%text = 'EINS';
display( text );

%text = 'eins';
display( text );
```

When you compile with PP(MACRO('RESCAN(ASIS)')), in the second display statement, the value of `text` is replaced by `eins`, but no further replacement occurs. This is because under RESCAN(ASIS), `eins` does not match the macro variable `eins`; the former is left as is while the latter is in uppercase. Hence the following text is generated:

```
DISPLAY( zwei );

DISPLAY( eins );
```

But when you compile with PP(MACRO('RESCAN(UPPER)')), in the second display statement, the value of `text` is replaced by `eins` , but further replacement does occur because under RESCAN(UPPER), `eins` does match the macro variable `eins` (both are in uppercase). Hence the following text is generated:

```
DISPLAY( zwei );

DISPLAY( zwei );
```

In summary, RESCAN(UPPER) ignores case while RESCAN(ASIS) does not.

You can set the default options for the macro preprocessor by using the `set IBM.PPMACRO` command.

Macro preprocessor options in the configuration file

You can set the default options for the macro preprocessor by using the **pliopt** or **macopt** attribute in the configuration file.

Macro preprocessor options environment variables

You can set the default options for the macro preprocessor by using the IBM.PPMACRO environment variable.

Macro preprocessor example

This example shows how to use the preprocessor to produce a source deck.

In the example shown in Figure 5, according to the value assigned to the preprocessor variable USE, the source statements represent either a subroutine (CITYSUB) or a function (CITYFUN).

The DSNAME used for SYSPUNCH specifies a source program library on which the preprocessor output will be placed. Normally compilation will continue and the preprocessor output will be compiled.

```
//OPT4#8  JOB
//STEP2 EXEC  IBMZC,PARM.PLI='MACRO,MDECK,NOCOMPILE,NOSYNTAX'
//PLI.SYSPUNCH DD  DSNAME=HPU8.NEWLIB(FUN),DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN  DD *
 /* GIVEN ZIP CODE, FINDS CITY                                        */
 %DCL USE CHAR;
 %USE = 'FUN'                  /* FOR SUBROUTINE, %USE = 'SUB' */ ;
 %IF USE = 'FUN' %THEN %DO;
 CITYFUN: PROC(ZIPIN) RETURNS(CHAR(16)) REORDER; /* FUNCTION         */
                       %END;
                 %ELSE %DO;
 CITYSUB: PROC(ZIPIN, CITYOUT) REORDER;     /* SUBROUTINE           */
   DCL CITYOUT CHAR(16);          /* CITY NAME                       */
                       %END;
   DCL (LBOUND, HBOUND) BUILTIN;
   DCL ZIPIN PIC '99999';        /* ZIP CODE                         */
   DCL 1 ZIP_CITY(7) STATIC,     /* ZIP CODE - CITY NAME TABLE       */
         2 ZIP PIC '99999' INIT(
                     95141, 95014, 95030,
                     95051, 95070, 95008,
                     0),           /* WILL NOT LOOK AT LAST ONE      */
         2 CITY CHAR(16) INIT(
                     'SAN JOSE', 'CUPERTINO', 'LOS GATOS',
                     'SANTA CLARA', 'SARATOGA', 'CAMPBELL',
                     'UNKNOWN CITY');  /* WILL NOT LOOK AT LAST ONE  */
   DCL I FIXED BIN(31);
   DO I = LBOUND(ZIP,1) TO             /* SEARCH FOR ZIP IN TABLE    */
          HBOUND(ZIP,1)-1              /* DON'T LOOK AT LAST ELEMENT */
          WHILE(ZIPIN ¬= ZIP(I));
   END;
 %IF USE = 'FUN' %THEN %DO;
   RETURN(CITY(I));                    /* RETURN CITY NAME           */
                       %END;
                 %ELSE %DO;
   CITYOUT=CITY(I);                    /* RETURN CITY NAME           */
                       %END;
 END;
```

Figure 5. Using the macro preprocessor to produce a source deck

SQL preprocessor

In general, the coding for your PL/I program is the same whether or not you want it to access a DB2 database. However, to retrieve, update, insert, and delete DB2 data and use other DB2 services, you must use SQL statements. You can use dynamic and static EXEC SQL statements in PL/I applications.

To communicate with DB2, you need to do the following:

- Code any SQL statements you need, delimiting them with **EXEC SQL**.
- Use the DB2 precompiler, or if using DB2 for z/OS Version 9 Release 1 or later, compile with the PL/I PP(SQL()) compiler option.

Before you can take advantage of the EXEC SQL support, you must have authority to access a DB2 system. Contact your local DB2 Database Administrator for your authorization.

Programming and compilation considerations

When you use the PL/I SQL preprocessor, the PL/I compiler handles your source program containing embedded SQL statements at compile time, without your having to use a separate precompile step. Although the use of a separate precompile step continues to be supported, use of the PL/I SQL preprocessor is recommended.

Interactive debugging with IBM Debug Tool is enhanced when you use the preprocessor because you see only the SQL statements while debugging (and not the generated PL/I source). However, you must have DB2 for z/OS Version 9 Release 1, or later to use the SQL preprocessor.

Using the preprocessor lifts some of the DB2 precompiler's restrictions on SQL programs. When you process SQL statements with the preprocessor, you can do the following:

- Use nested SQL INCLUDE statements.
- Use fully-qualified names for structured host variables.
- Include SQL statements at any level of a nested PL/I program, instead of in only the top-level source file.
- Use the SQL TYPE attribute anywhere you can specify a PL/I data type. All such attributes can be factored and can be used in structure elements, in arrays, and with any storage class like BASED.
- Use variables declared with the LIKE attribute as host variables.

All PL/I statements must be syntactically correct, because the SQL preprocessor scans the source looking for EXEC SQL statements, DECLARE statements, and statements that delimit blocks of declarations. If a statement is coded incorrectly, this might mislead the preprocessor when the preprocessor looks for the END statement to a BEGIN, DO, PACKAGE, PROCEDURE, or SELECT statement, and that can cause the preprocessor to be unable to resolve some host variable references correctly. To help identify such incorrect code, the SQL preprocessor flags errors as follows:

- Left parentheses without matching right parentheses
- SELECT statements that do not end with a semicolon
- IF statements that do not contain a THEN keyword
- Statements that start with an invalid symbol

If your source contains both MACRO and SQL statements:

- You might invoke the SQL preprocessor without the INCONLY option only after invoking the MACRO preprocessor. In that case, EXEC SQL statements are allowed anywhere executable PL/I statements are allowed.
- You might invoke the SQL preprocessor with the INCONLY option before invoking the MACRO preprocessor, but in that case, EXEC SQL statements are allowed only if they are immediately preceded by a semicolon (with possibly intervening whitespace and comments, of course).

The SQL preprocessor supports DBCS in the same manner as the PL/I compiler does. When the GRAPHIC PL/I compiler option is in effect, some source language elements can be written in DBCS and SBCS characters. In particular, you can use DBCS characters in the source program in following places:

- Inside comments
- As part of statement labels and identifiers
- In G or M literals

The following restrictions apply to the use of PL/I built-in functions, compiler options, and statements when you program and compile SQL statements:

- When EXEC SQL statements are translated to PL/I, the following built-in functions might be included in the generated code. If you use any of the following built-in functions as the names of elements in structures, you must also explicitly declare them as BUILTIN:
  - ADDR
  - LENGTH
  - MAXLENGTH
  - PTRVALUE
  - SYSNULL
- You must not use PL/I type functions, such as BIND(: *t*,*p*:), in EXEC SQL statements.
- When compiling with the preprocessor, you must not use these compiler options:
  - DFT(ASCII)
  - DFT(IEEE)
- Do not use DECLARE STATEMENT statements in SQL queries, because the PL/I preprocessor always ignores these statements.

Compiling with the SQL preprocessor option generates a DB2 database request module (DBRM) along with the usual PL/I compiler output such as object module and listing. As input to the DB2 bind process, the DBRM data set contains information about the SQL statements and host variables in the program. Not all of the information in the DBRM is important in terms of the bind or runtime processing, however. For example, if the HOST value in the DBRM specifies a language other than PL/I, there is no reason to be concerned. All this means that the other language is selected as the installation default for the HOST value, which does not affect the bind or runtime processing of your program.

If the EMPTYDBRM option is in effect and if the source meets one of the following conditions, the preprocessor issues a message indicating that no statements required translation, and does not create a DBRM:

- The source contains no EXEC SQL statements.

- The source contains only EXEC SQL INCLUDE statements other than EXEC SQL INCLUDE SQLCA and EXEC SQL INCLUDE SQLDA.

The PL/I compiler listing includes the error diagnostics (such as syntax errors in the SQL statements) that the preprocessor generates. The listing of the EXEC SQL statement is displayed in a readable format that is similar to the original source.

To use the preprocessor, you need to do the following:

- Specify the following option when you compile your program:

  ```
  PP(SQL('options'))
  ```

  This compiler option indicates that you want the compiler to invoke the integrated SQL preprocessor. Specify a list of SQL processing options in the parenthesis after the SQL keyword. The options can be separated by a comma or by a space and the list of options must be enclosed in quotation marks (single or double, as long as they match).

  For example, `PP(SQL('DATE(USA),TIME(USA)')` tells the preprocessor to use the USA format for both DATE and TIME data types.

  In addition, for LOB support you must specify the following option:

  ```
  LIMITS( FIXEDBIN(31,63)  FIXEDDEC(15,31) )
  ```

  An alternative way to specify SQL preprocessor options is to use the PPSQL compiler option. For information about how to use it, see "PPSQL" on page 64.
- Include DD statements for the following data sets in the JCL for your compile step:
  - DB2 load library (*prefix*.SDSNLOAD)

    The SQL preprocessor calls DB2 modules to do the SQL statement processing. Therefore, you need to include the name of the DB2 load library data set in the STEPLIB concatenation for the compile step.
  - Library for SQL INCLUDE statements

    If your program contains SQL INCLUDE *member-name* statements that specify secondary input to the source program, you need to include the name of the data set that contains *member-name* in the SYSLIB concatenation for the compile step.
  - DBRM library

    The compilation of the PL/I program generates a DB2 database request module (DBRM), and the DBRMLIB DD statement is required to designate the data set to which the DBRM is written.

    The DBRMLIB DD statement must name a data set that can be opened and closed more than once during the compilation.

  For example, you might have the following lines in your JCL:

  ```
  //STEPLIB  DD  DSN=DSNA10.SDSNLOAD,DISP=SHR
  //SYSLIB   DD  DSN=PAYROLL.MONTHLY.INCLUDE,DISP=SHR
  //DBRMLIB  DD  DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
  ```

SQL preprocessor options

This section describes the options that the SQL preprocessor supports.

Two groups of options can be passed to the SQL preprocessor: those handled by the PL/I SQL preprocessor and those handled by the DB2 coprocessor. You must specify these options in the options string of the PP(SQL('*option-list*')) option, and the options can be intermingled in the options string.

When you specify SQL preprocessor options, the list of options must be enclosed in a pair of quotation marks. For example, to specify the CCSID0 option, you must specify PP(SQL('CCSID0')).

Table 8 lists the PL/I SQL preprocessor options with abbreviations (if any) and the IBM-supplied default values. This table uses a vertical bar (|) to separate mutually exclusive options, and brackets ([ ]) to indicate that you can sometimes omit the enclosed option.

For more information about DB2 coprocessor options, see the *DB2 for z/OS Application Programming and SQL Guide*.

Table 8. SQL preprocessor options and IBM-supplied defaults

SQL Preprocessor option	Abbreviated name	z/OS default
CCSID0 | NOCCSID0	-	CCSID0
CODEPAGE | NOCODEPAGE	-	NOCODEPAGE
DEPRECATE(STMT([EXPLAIN | GRANT | REVOKE | SET_CURRENT_SQLID]))	-	DEPRECATE(STMT())
EMPTYDBRM | NOENTRYDBRM	-	NOENTRYDBRM
HOSTCOPY | NOHOSTCOPY	-	HOSTCOPY
INCONLY | NOINCONLY	-	NOINCONLY
WARNDECP | NOWARNDECP	-	NOWARNDECP

CCSID0

The CCSID0 option specifies that no host variable other than WIDECHAR is to be assigned a CCSID value by the PL/I SQL preprocessor.

The NOCCSID0 option allows host variables to be assigned a CCSID value by the PL/I SQL preprocessor.

If your program updates FOR BIT DATA columns with a data type that is not BIT data, choose CCSID0. CCSID0 informs DB2 that the host variable is not associated with a CCSID, allowing the assignment to be made. Otherwise, if a host variable that is associated with a CCSID that is not BIT data is assigned to a FOR BIT DATA column, a DB2 error occurs.

WIDECHAR is always assigned a CCSID value of 1200.

For compatibility with older PL/I programs that used the DB2 precompiler, enable CCSID0.

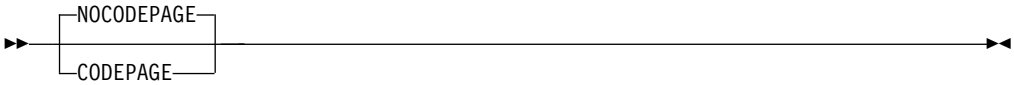
CCSID0 and NOCCSID0 are mutually exclusive options.

The default is CCSID0.

CODEPAGE

When the CODEPAGE option is in effect, the compiler CODEPAGE option is always used as the CCSID for SQL host variables of character type.

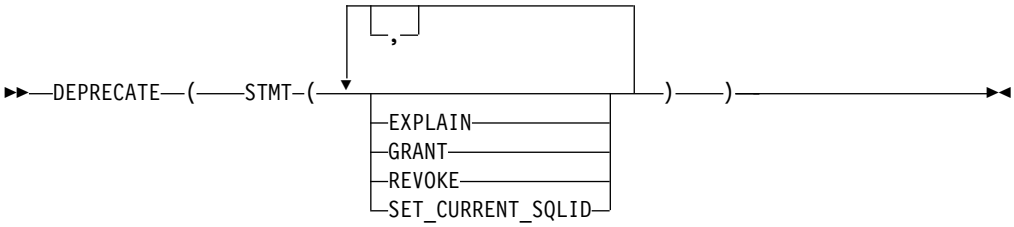
When the NOCODEPAGE option is in effect, the compiler CODEPAGE option is used as the CCSID for SQL host variables of character type only if the SQL preprocessor option NOCCSID0 is also in effect.



The default is NOCODEPAGE.

DEPRECATE

The DEPRECATE option indicates that the preprocessor flags the specified statements as deprecated.



STMT
Specifies a list of statements that the preprocessor should flag as deprecated. The list can be empty.

EXPLAIN
The EXPLAIN SQL statement.

GRANT
The GRANT SQL statement.

REVOKE
The REVOKE SQL statement.

SET_CURRENT_SQLID
The SET CURRENT SQLID SQL statement.

The default is DEPRECATE(STMT()).

EMPTYDBRM

The EMPTYDBRM option specifies that the SQL preprocessor always creates a DBRM even if the code contains no EXEC SQL statements. However, the SQL preprocessor does not create a DBRM when the INCONLY option is invoked.

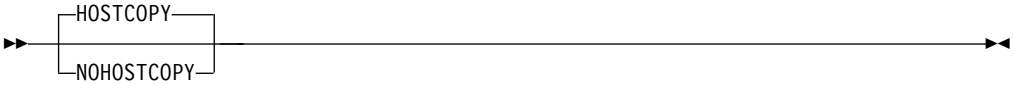
The NOENTRYDBRM option specifies that the SQL preprocessor does not create a DBRM.

The default is NOENTRYDBRM.

HOSTCOPY

The HOSTCOPY option determines if the SQL preprocessor, when running under LP(64), generates code to copy host variables to and from below-the-bar storage around each EXEC SQL statement.

If the NOHOSTCOPY is specified, the SQL preprocessor does not generate this code. However, it is then the user's responsibility to ensure that all host variables reside in below-the-bar storage (for example, by making them BASED variables whose base pointers are obtained via the ALLOC31 built-in function).



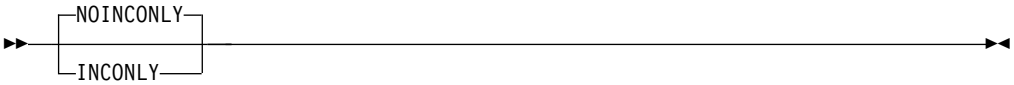
The default is HOSTCOPY.

The HOSTCOPY option is ignored under LP(32).

INCONLY

The INCONLY option specifies that the SQL preprocessor should process only EXEC SQL INCLUDE statements except for `includes` of the SQL communication area (SQLCA) and the SQL descriptor area (SQLDA). No code is generated by the SQL preprocessor when this option is in effect.

The NOINCONLY option specifies that the SQL preprocessor should process all statements, not only EXEC SQL INCLUDE statements.



When you specify the INCONLY option, the compiler does not produce the SQL options listing, because all other options are ignored under INCONLY.

The INCONLY option and the NOINCONLY option are mutually exclusive.

For compatibility, the default is NOINCONLY.

WARNDECP

When the WARNDECP option is in effect, the preprocessor issues a warning message if the compilation uses the DSNHDECP module supplied with DB2.

When the NOWARNDECP option is in effect, no warning message is issued.

The default is NOWARNDECP.

PL/I-specific notes for SQL processor options

The topic describes some rules that you must follow when specifying SQL processor options FLOAT, ONEPASS, and STDSQL.

When you specify the following SQL processor options for the PL/I compiler, the following rules apply:

FLOAT
An error message is issued if the FLOAT option is different from the PL/I DEFAULT(HEXADEC | IEEE) option.

ONEPASS | TWOPASS
Under the option ONEPASS, you must declare all host variables before they are used by any SQL statement.

STDSQL
Under the option STDSQL(YES), you must declare all host variables between SQL BEGIN DECLARE SECTION and SQL END DECLARE SECTION statements.

Coding SQL statements in PL/I applications

You can code SQL statements in your PL/I applications by using the language defined in *DB2 UDB for z/OS SQL Reference*. This section describes specific requirements for your SQL code.

Defining the SQL communications area

A PL/I program that contains SQL statements must include either an SQLCODE variable (if the STDSQL(86) preprocessor option is used) or an SQL communications area (SQLCA).

As shown in Figure 6 on page 130, part of an SQLCA consists of an SQLCODE variable and an SQLSTATE variable.

- The SQLCODE value is set by the Database Services after each SQL statement is executed. An application can check the SQLCODE value to determine whether the last SQL statement was successful.
- The SQLSTATE variable can be used as an alternative to the SQLCODE variable when the compiler analyzes the result of an SQL statement. Like the SQLCODE variable, the SQLSTATE variable is set by the Database Services after each SQL statement is executed.

To include the SQLCA declaration, use the EXEC SQL INCLUDE statement:

```
exec sql include sqlca;
```

The SQLCA structure must not be defined within an SQL declare section. The scope of the SQLCODE and SQLSTATE declarations must include the scope of all SQL statements in the program.

```
Dcl
  1 Sqlca,
    2 sqlcaid      char(8),           /* Eyecatcher = 'SQLCA   '   */
    2 sqlcabc      fixed binary(31),  /* SQLCA size in bytes = 136 */
    2 sqlcode      fixed binary(31),  /* SQL return code           */
    2 sqlerrmc     char(70) var,      /* Error message tokens      */
    2 sqlerrp      char(8),           /* Diagnostic information    */
    2 sqlerrd(0:5) fixed binary(31),  /* Diagnostic information    */
    2 sqlwarn,                        /* Warning flags             */
      3 sqlwarn0   char(1),
      3 sqlwarn1   char(1),
      3 sqlwarn2   char(1),
      3 sqlwarn3   char(1),
      3 sqlwarn4   char(1),
      3 sqlwarn5   char(1),
      3 sqlwarn6   char(1),
      3 sqlwarn7   char(1),
    2 sqlext,
      3 sqlwarn8   char(1),
      3 sqlwarn9   char(1),
      3 sqlwarna   char(1),
      3 sqlstate   char(5);       /* State corresponding to SQLCODE */
```

Figure 6. The PL/I declaration of SQLCA

Defining SQL descriptor areas

Unlike the SQLCA, there can be more than one SQL descriptor area (SQLDA) in a program, and an SQLDA can have any valid name.

The following statements require an SQLDA:

PREPARE *statement-name* INTO *descriptor-name* FROM *host-variable*
EXECUTE...USING DESCRIPTOR *descriptor-name*
FETCH...USING DESCRIPTOR *descriptor-name*
OPEN...USING DESCRIPTOR *descriptor-name*
DESCRIBE *statement-name* INTO *descriptor-name*

To include an SQLDA, use the EXEC SQL INCLUDE statement:

```
exec sql include sqlda;
```

The SQLDA must not be defined within an SQL declare section.

```
Dcl
  1 Sqlda based(Sqldaptr),
    2 sqldaid     char(8),          /* Eye catcher = 'SQLDA   ' */
    2 sqldabc     fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
    2 sqln        fixed binary(15), /* Number of SQLVAR elements*/
    2 sqld        fixed binary(15), /* # of used SQLVAR elements*/
    2 sqlvar(Sqlsize refer(sqln)),  /* Variable Description     */
      3 sqltype   fixed binary(15), /* Variable data type       */
      3 sqllen    fixed binary(15), /* Variable data length     */
      3 sqldata   pointer,          /* Pointer to variable data value*/
      3 sqlind    pointer,          /* Pointer to Null indicator*/
      3 sqlname   char(30) var ;    /* Variable Name            */

Dcl
  1 Sqlda2 based(Sqldaptr),
    2 sqldaid2     char(8),          /* Eye catcher = 'SQLDA   ' */
    2 sqldabc2     fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
    2 sqln2        fixed binary(15), /* Number of SQLVAR elements*/
    2 sqld2        fixed binary(15), /* # of used SQLVAR elements*/
    2 sqlvar2(Sqlsize refer(sqln2)), /* Variable Description     */
      3 sqlbiglen,
        4 sqllongl fixed binary(31),
        4 sqlrsvdl fixed binary(31),
      3 sqldatal   pointer,
      3 sqltname   char(30) var;

dcl Sqlsize    fixed binary(15);     /* number of sqlvars (sqln) */
dcl Sqldaptr   pointer;
dcl Sqltripled char(1) value('3');
dcl Sqldoubled char(1) value('2');
dcl Sqlsingled char(1) value(' ');
```

Figure 7. The PL/I declaration of an SQL descriptor area

Embedding SQL statements

The first statement of your program must be a PROCEDURE or PACKAGE statement. You can add any SQL statement to your program wherever executable statements can appear.

You can also add the following SQL statements in a PACKAGE and outside of any procedure:

- EXEC SQL BEGIN DECLARE SECTION
- EXEC SQL END DECLARE SECTION
- EXEC SQL DECLARE if no executable code needs to be generated for the statement
- EXEC SQL INCLUDE

Each SQL statement must begin with `EXEC (or EXECUTE) SQL` and end with a semicolon (;).

For example, an UPDATE statement might be coded as follows:

```
exec sql update DSN8A10.DEPT
set   Mgrno  = :Mgr_Num
where Deptno = :Int_Dept;
```

Comments:

In addition to SQL statements, comments can be included in embedded SQL statements wherever a blank is allowed.

If a comment appears inside a SQL statement, the forward slash (/) that closes the comment is shown as a greater than sign (>) in the listing. The following example illustrates how the compiler will show a given SQL statement in the source listing.

The following example SQL statement contains comments:

```
exec sql insert into table /* some text */ values(:data);
```

The compiler will show it in the source listing as follows:

```
/*$*$*$
 exec sql insert into table /* some text *> values(:data)
 $*$*$*/
```

SQL style comments ('--') are supported when embedded in SQL statements.

Continuation for SQL statements:

The line continuation rules for SQL statements are the same as those for other PL/I statements.

Including code: You can include SQL statements or PL/I host variable declaration statements by placing the following SQL statement in the source code. Place it at the point where the statements are to be embedded.

```
exec sql include member;
```

Margins: You must code SQL statements in columns *m* through *n*, where *m* and *n* are specified in the MARGINS(*m*,*n*) compiler option.

Names: You can use any valid PL/I variable name for a host variable. The length of a host variable name must not exceed the value *n* specified in the LIMITS(NAME(*n*)) compiler option.

Statement labels: With the exception of the END DECLARE SECTION statement and the INCLUDE text-file-name statement, executable SQL statements, like PL/I statements, can have a label prefix.

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be a label in the PL/I source code and must be within the scope of any SQL statements affected by the WHENEVER statement.

Using host variables

All host variables used in SQL statements must be explicitly declared, and all host variables within an SQL statement must be preceded by a colon (:).

Subscripts must not be used in host variable references.

The following topics describe the details of using host variables:

- "Using arrays as host variables" on page 133
- "Declaring host variables" on page 133
- "Declaring scalar host variables" on page 134
- "Determining equivalent SQL and PL/I data types" on page 136

• "Determining compatibility of SQL and PL/I data types" on page 139

Using arrays as host variables: You can use an array as a host variable only in the following two ways:

- As an array of indicator variables for a host structure
- As an array of host variables when used in any of the following statements:
  - A FETCH statement for a multiple row fetch
  - An INSERT statement with a multiple row insert
  - A multiple row MERGE statement

All such arrays must be one-dimensional, have the CONNECTED attribute, and have constant bounds.

All other use of arrays as host variables is invalid.

Declaring host variables:

Host variable declarations can be made at the same place as regular PL/I variable declarations.

Only a subset of valid PL/I declarations are recognized as valid host variable declarations.

The SQL preprocessor parses the DEFINE ALIAS, DEFINE ORDINAL, and DEFINE STRUCTURE statements. This means that if a DEFINE ALIAS statement defines a PL/I type that can be used in SQL statements, a variable declared with that type can also be used in SQL statements.

The preprocessor does not use the data attribute defaults specified in the PL/I DEFAULT statement. If the declaration for a variable is not recognized, any statement that references the variable might result in the message:

```
'The host variable token ID is not valid'
```

To use a structure that is declared with LIKE or an element of this structure as a host variable, the declaration for the LIKE object must be visible to the SQL preprocessor. For example, the LIKE object must not be in a %INCLUDE file that has not been included.

You can use restricted expressions in host variable declarations to define the bounds of an array or the length of a string as long as the expression has one of the following forms:

- A prefix operator applied to an expression where the expression can be collapsed to an integer
- An add operator or a subtract operator applied to two expressions where both expressions can be collapsed to integers
- A multiply operator applied to two expressions where both expressions can be collapsed to integers
- A reference to a named constant where the reference can be collapsed to an integer
- One of these built-in functions: INDICATORS, HBOUND, HBOUNDACROSS, LENGTH, and MAXLENGTH
- A number that is an integer

Although you can use a named constant to define the bounds and lengths of a host variable, you cannot use a named constant itself as a host variable except if both of the following conditions apply:

- DB2 allows a simple, unnamed constant at that place in the EXEC SQL statement.
- The named constant has any of these attributes:
  - CHARACTER, in which case the VALUE attribute of the named constant must specify a character string.
  - FIXED, in which case the VALUE attribute of the named constant must specify a decimal number or an expression that can be reduced, with the same restrictions as above, to an integer constant.

Only the names and data attributes of the variables are used by the preprocessor; the alignment, scope, and storage attributes are ignored.

Declaring scalar host variables: You must declare a scalar host variable with one of the following data attributes:

CHARACTER, GRAPHIC, or WIDECHAR

Host variables that are declared with the CHARACTER, GRAPHIC, or WIDECHAR attributes are called string host variables. The following restrictions apply to the string host variable:

- It must have either the NONVARYING or VARYING attribute.
- If it has the VARYING attribute, it must have the NATIVE attribute.

FIXED BINARY, FIXED DECIMAL, or FLOAT

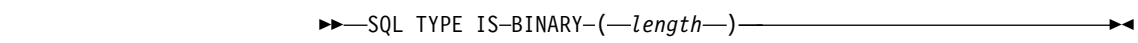
Host variables that are declared with the FIXED and BINARY, FIXED and DECIMAL, or FLOAT attributes are called numeric host variables. The following restrictions apply to the numeric host variable:

- It must have the REAL attribute.
- If it has the FIXED and BINARY attributes, it must have the SIGNED and NATIVE attributes, a zero scale factor, and a precision greater than 7.
- If it has the FIXED and DECIMAL attributes, it must have a nonnegative scale factor that is smaller than its precision.
- If it has the FLOAT and DECIMAL attributes, it must have a precision that is less than 17 unless the FLOAT(DFP) option is in effect.
- If it has the FLOAT and BINARY attributes, it must have a precision that is less than 54.

SQL TYPE

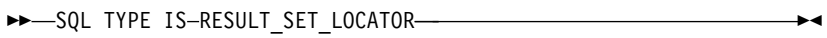
Host variables that are declared with the SQL TYPE attribute are called SQL TYPE host variables. The attribute specification must conform to one of the following syntax diagrams:

BINARY



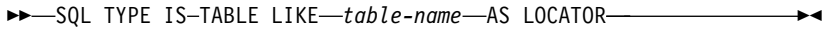
VARBINARY

Result set locator

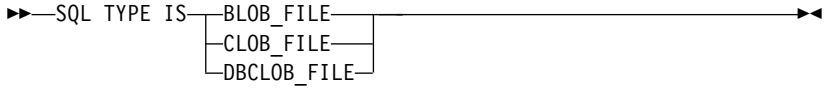


ROWID

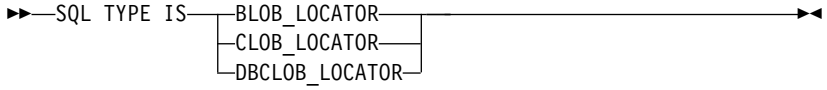
Table locator



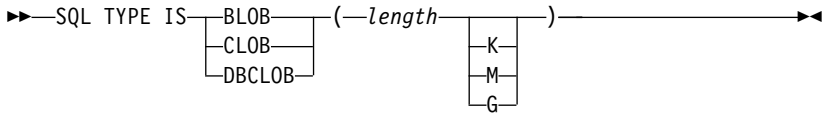
LOB file reference



LOB locator



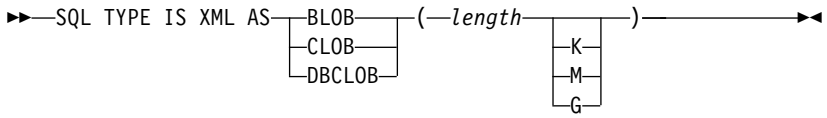
LOB variable



BLOB
You can also use **BINARY LARGE OBJECT** as an alternative to **BLOB**.

CLOB
You can also use either **CHARACTER LARGE OBJECT** or **CHAR LARGE OBJECT** as an alternative to **CLOB**.

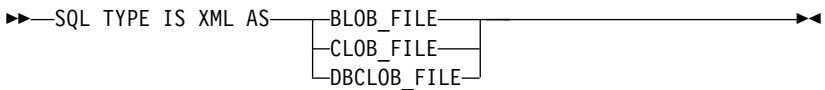
XML LOB variable



BLOB
You can also use **BINARY LARGE OBJECT** as an alternative to **BLOB**.

CLOB
You can also use either **CHARACTER LARGE OBJECT** or **CHAR LARGE OBJECT** as an alternative to **CLOB**.

XML file reference



The following constant declarations are generated by the SQL preprocessor. You can use them to set the file option variable when you use the file reference host variables:

```
DCL SQL_FILE_READ      FIXED BIN(31) VALUE(2);
DCL SQL_FILE_CREATE    FIXED BIN(31) VALUE(8);
DCL SQL_FILE_OVERWRITE FIXED BIN(31) VALUE(16);
DCL SQL_FILE_APPEND    FIXED BIN(31) VALUE(32);
```

Determining equivalent SQL and PL/I data types: The base SQLTYPE and SQLLEN of host variables are determined according to Table 9 and Table 10 on page 137. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

To determine the PL/I data type that is equivalent to a given SQL data type, you can use Table 11 on page 137 and Table 12 on page 138.

Table 9. SQL data types generated from PL/I declarations

PL/I data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
BIN FIXED(p), 7 < p <= 15	500	2	SMALLINT
BIN FIXED(p), 15 < p <= 31	496	4	INTEGER
BIN FIXED(p), 31 < p <= 63	492	8	BIGINT
DEC FIXED(p,s), 0<=p<=15 and 0<=s<=p	484	p (byte 1) s (byte 2)	DECIMAL(p,s)
BIN FLOAT(p), 1 ≤ p ≤ 21	480	4	REAL or FLOAT(n) 1<=n<=21
BIN FLOAT(p), 22 ≤ p ≤ 53	480	8	DOUBLE PRECISION or FLOAT(n), 22<=n<=53
Under FLOAT(NODFP):			
DEC FLOAT(p), 1 ≤ p ≤ 6	480	4	FLOAT (single precision)
DEC FLOAT(p), 7 ≤ p ≤ 16	480	8	FLOAT (double precision)
Under FLOAT(DFP):			
DEC FLOAT(p), 1 ≤ p ≤ 7	996	4	DECFLOAT (single precision)
DEC FLOAT(p), 7 ≤ p ≤ 16	996	8	DECFLOAT (double precision)
DEC FLOAT(p), 16 ≤ p ≤ 34	996	16	DECFLOAT (extended decimal)
CHAR(n)	452	n	CHAR(n)
CHAR(n) VARYING	448	n	VARCHAR(n)
GRAPHIC(n), 1 ≤ n ≤ 127	468	n	GRAPHIC(n)
GRAPHIC(n) VARYING	464	n	VARGRAPHIC(n)

Table 10. SQL data types generated from SQL TYPE declarations

PL/I data type	SQLTYPE of host variable	SQLLEN of host variable	SQL data type
SQL TYPE IS BLOB(*n*) 1<*n*<2147483647	404	*n*	BLOB(*n*)
SQL TYPE IS CLOB(*n*) 1<*n*<2147483647	408	*n*	CLOB(*n*)
SQL TYPE IS DBCLOB(*n*) 1<*n*<1073741823 (2)	412	*n*	DBCLOB(*n*) (2)
SQL TYPE IS ROWID	904	40	ROWID
SQL TYPE IS VARBINARY(*n*) 1<*n*<32704	908	*n*	VARBINARY(*n*)
SQL TYPE IS BINARY(*n*) 1<*n*<255	912	*n*	BINARY(*n*)
SQL TYPE IS BLOB_FILE	916	267	BLOB File Reference (1)
SQL TYPE IS CLOB_FILE	920	267	CLOB File Reference (1)
SQL TYPE IS DBCLOB_FILE	924	267	DBCLOB File Reference (1)
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB Locator (1)
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB Locator (1)
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB Locator (1)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result Set Locator
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table Locator (1)

Notes:
1. Do not use this data type as a column type.
2. *n* is the number of double-byte characters.

Table 11. SQL data types mapped to PL/I declarations

SQL data type	PL/I equivalent	Notes®
SMALLINT	BIN FIXED(15)	
INTEGER	BIN FIXED(31)	
BIGINT	BIN FIXED(63)	
DECIMAL(p,s)	DEC FIXED(p) or DEC FIXED(p,s)	p = precision and s = scale; 1 ≤ p ≤ 31 and 0 ≤ s ≤ p
Under FLOAT(NODFP):		
REAL or FLOAT(n)	BIN FLOAT(p) or DEC FLOAT(m)	1 ≤ p ≤ 21 and 1 ≤ m ≤ 6
DOUBLE PRECISION, DOUBLE, or FLOAT(n)	BIN FLOAT(p) or DEC FLOAT(m)	22 ≤ p ≤ 53 and 7 ≤ m ≤ 16
Under FLOAT(DFP):		
DECFLOAT	DEC FLOAT(m)	Short Decimal Float 1 ≤ m ≤ 7

DECFLOAT	DEC FLOAT(m)	Long Decimal Float 7 ≤ m ≤ 16
DECFLOAT	DEC FLOAT(m)	Extended Decimal Float 16 ≤ m ≤ 34

CHAR(n)	CHAR(n)	1 ≤ n ≤ 32767
VARCHAR(n)	CHAR(n) VAR	
GRAPHIC(n)	GRAPHIC(n)	*n* is a positive integer that refers to the number of double-byte characters, not to the number of bytes; 1 ≤ n ≤ 16383
VARGRAPHIC(n)	GRAPHIC(n) VAR	*n* is a positive integer that refers to the number of double-byte characters, not to the number of bytes; 1 ≤ n ≤ 16383
DATE	CHAR(n)	*n* must be at least 10.
TIME	CHAR(n)	*n* must be at least 8.
TIMESTAMP	CHAR(n)	*n* must be at least 26.

Table 12. SQL data types mapped to SQL TYPE declarations

SQL data type	PL/I equivalent	Notes
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB file reference	SQL TYPE IS BLOB_FILE	Use this data type only as a reference to a BLOB file. Do not use this data type as a column type.
CLOB file reference	SQL TYPE IS CLOB_FILE	Use this data type only as a reference to a CLOB file. Do not use this data type as a column type.
DBCLOB file reference	SQL TYPE IS DBCLOB_FILE	Use this data type only as a reference to a DBCLOB file. Do not use this data type as a column type.
BLOB(*n*)	SQL TYPE IS BLOB(*n*)	1<*n*<2147483647
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	1<*n*<2147483647
DBCLOB(*n*)	SQL TYPE IS DBCLOB(*n*)	*n* is the number of double-byte characters. 1<*n*<1073741823
ROWID	SQL TYPE IS ROWID	
XML AS	SQL TYPE IS XML AS ...	Used to describe an XML version of a BLOB, CLOB, DBCLOB, BLOB_FILE, CLOB_FILE, or DBCLOB_FILE

Determining compatibility of SQL and PL/I data types: PL/I host variables in SQL statements must be type-compatible with the columns that use them:

- Numeric data types are compatible with each other. A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a PL/I host variable of BIN FIXED(15), BIN FIXED(31), DECIMAL(*p*,*s*), BIN FLOAT(*n*) where *n* is in the range 22 - 53, or DEC FLOAT(*m*) where *m* is in the range 7 - 16.
- Character data types are compatible with each other. A CHAR or VARCHAR column is compatible with a fixed-length or varying-length PL/I character host variable.
- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.

  When necessary, the Database Manager automatically converts a fixed-length character string to a varying-length string or a varying-length string to a fixed-length character string.

Using host structures

A PL/I host structure name can be a structure name with members that are not structures or unions.

In the following example, B is the name of a host structure consisting of the scalars C1 and C2.

```
dcl 1 A,
      2 B,
        3 C1 char(...),
        3 C2 char(...);
```

Host structures are limited to two levels. A host structure can be thought of as a named collection of host variables.

Each leaf element of a host structure must have one of the following valid host data attributes as discussed in "Declaring host variables" on page 133:

- CHARACTER, GRAPHIC, or WIDECHAR
- FIXED BINARY, FIXED DECIMAL, or FLOAT
- SQL TYPE

Using indicator variables

An indicator variable is a two-byte integer (BIN FIXED(15)), an array of two-byte integers, or a structure that contains only two-byte integers (or arrays thereof).

On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to indicate that a null value should be assigned.

A structure can be used as an indicator variable only when the associated host variable is also a structure.

Indicator variables are declared in the same way as host variables and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

The SQL Preprocessor does not require that indicator arrays have a lower bound of one.

An indicator variable must have the attribute REAL NATIVE SIGNED FIXED BIN(15).

The following example shows how to declare variables for the statement shown in Figure 8.

```
exec sql fetch Cls_Cursor into :Cls_Cd,
                               :Day :Day_Ind,
                               :Bgn :Bgn_Ind,
                               :End :End_Ind;
```

Figure 8. SQL statement containing indicator variables

You can declare variables as follows:

```
exec sql begin declare section;
dcl Cls_Cd    char(7);
dcl Day       bin fixed(15);
dcl Bgn       char(8);
dcl End       char(8);
dcl (Day_Ind, Bgn_Ind, End_Ind)   bin fixed(15);
exec sql end declare section;
```

Host structure example

This example shows the declaration of a host structure and an indicator array followed by an SQL statement that can be used to retrieve the data into the host structure.

```
dcl 1 games,
      5 sunday,
        10 opponents char(30),
        10 gtime     char(10),
        10 tv        char(6),
        10 comments  char(120) var;
dcl indicator(4) fixed bin (15);


exec sql
     fetch cursor_a
     into :games.sunday:indicator;
```

Manipulating LOB data

LOBS, CLOBS, and BLOBS can be as large as 2,147,483,647 bytes long (2 Gigabytes). Double Byte CLOBS can be 1,073,741,823 characters long (1 Gigabyte). To use large object (LOB) data from a DB2 table, use techniques like LOB locators and LOB file references to manipulate the data while the data is still in the database.

Declaring a host variable to hold all of the LOB data can be inefficient or impractical, because this requires your program to allocate large amounts of storage and requires DB2 to move large amounts of data. Therefore, it is recommended that you use the following techniques to manipulate LOB data:

- LOB locators

  Using LOB locators, you can manipulate LOB data without moving the LOB data into host variables. By using LOB locators, you need much smaller amounts of memory for your programs.
- LOB file references

  You can use LOB file reference variables to import or export data between a LOB column and an external file outside the DB2 system.

For more information about these techniques to minimize the moving around of large pieces of data, see the *DB2 for z/OS Application Programming and SQL Guide*, or the IBM Redbooks® publication, *LOBs with DB2 for z/OS: Stronger and Faster*.

See pliclob sample program for an example of how to manipulate CLOBs in a PL/I and DB2 environment.

LOB locators

You can use LOB Locators to avoid materialization of the LOB data and all the underlying activities associated with it.

The benefits of using LOB locators are listed as follows:

- Saving storage when manipulating LOBs with LOB locators
- Manipulating data without retrieving it from the database
- Avoiding the use of large amounts of storage to hold the LOB
- Avoiding the time and resource expenditures for moving large pieces of data thereby improving performance

LOB locators are especially useful under the following circumstances:

- When you need only a small part of the LOB
- When you do not have enough memory for the entire LOB
- When performance is important
- In a client or server environment to avoid moving data over the network from one system to another

The following code example is from pliclob sample program. The sample program uses LOB locators to identify and manipulate sections of the `resume` CLOB from the `dsn8a10.emp_photo_resume` DB2 V10 table. (The numbers that precede each line are not part of the program, but are used in the explanation after the program.)

```
1.   dcl hv_loc_resume  sql type is clob_locator;
2.    exec sql
3.       select resume into :hv_loc_resume
4.       from dsn8a10.emp_photo_resume
5.       where empno = :hv_empno;
6.
7.    exec sql
8.       set :start_resume = (posstr(:hv_loc_resume, 'Resume:'));
```

In lines 2 - 5, LOB locator `hv_loc_resume` is set to the location of the resume of the employee number `hv_empno` in the `emp_photo_resume` table. In lines 7 - 8, the *start_resume* host variable is set to the beginning of the `'Resume:'` section of the `resume`. Then you can start manipulating the `resume` data while the resume is still in the data base.

LOB file reference variables

You can use LOB file reference variables to import or export data between a LOB column and an external file outside the DB2 system.

The benefits of using LOB file reference variables are listed as follows:

- Uses less processing time than moving LOB data with a host variable. The movement of the data would not be overlapped with any DB2 processing or network transfer time.
- Uses less application storage. LOB data is moved directly from DB2 to a file and is not materialized in the memory of the application.

Host structure example

The following code example is from pliclob sample program. The sample program uses LOB file references to create a new, trimmed down version of the resume in an external file. (The numbers that precede each line are not part of the program, but are used in the explanation after the program.)

```
1.   dcl hv_clob_file   sql type is clob_file;
2.   name_string = '/SYSTEM/tmp/pliclob2.txt';
3.   hv_clob_file.sql_lob_file_name_len = length(name_string);
4.   hv_clob_file.sql_lob_file_name     = name_string;
5.   hv_clob_file.sql_lob_file_options  = ior(sql_file_overwrite);
6.
7.   exec sql
8.    values ( substr(:hv_loc_resume,:start_resume,
9.                    :start_pers_info-:start_resume)
10.          || substr(:hv_loc_resume,:start_work_hist,
11.                     :start_interests-:start_work_hist)
12.          )
13.    into :hv_clob_file;
```

The host variable *hv_clob_file* is declared as a LOB file reference. In lines 2 - 4, the file name field of the LOB file reference is set to the fully qualified file name and file name length is set to its length. The overwrite option is set so any existing file is overwritten (line 5). For details of these and other file options, see the *DB2 for z/OS Application Programming and SQL Guide*.

Next the SQL VALUES statement is used to concatenate the resume name and work history sections of the resume directly into the hv_clob_file LOB file reference, as in lines 8 - 13.

Example: pliclob sample program

This sample PL/I program shows how to manipulate CLOBs in a PL/I and DB2 environment.

You must have the DB2 supplied sample database installed for this program to run properly. This sample assumes DB2 V10 and table dsn8a10.emp_photo_resume. If you use a different version of DB2, you must change the table reference.

Notes:

- When you use the LOB locators and LOB file reference variables, the resume CLOB is still within the database and not in memory or storage.
- The format of the resume CLOB in the database remains unchanged, with the reformatting of the resume taking place only in the second file that was written out.

```
pliclob: procedure options(main);
 display('begin pliclob');
 exec sql include sqlca;

 dcl hv_empno       char(06);
 dcl name_string    char(256) var;
 dcl hv_resume      sql type is clob(50k);
 dcl hv_clob_file   sql type is clob_file;
 dcl hv_loc_resume  sql type is clob_locator;

 dcl start_resume     fixed bin(31);
 dcl start_pers_info  fixed bin(31);
 dcl start_dept_info  fixed bin(31);
 dcl start_education  fixed bin(31);
 dcl start_work_hist  fixed bin(31);
 dcl start_interests  fixed bin(31);

 /* Extract resume CLOB for employee '000130' into a file in z/OS  */
 /* UNIX file system. The contents of this file shows the initial  */
 /* format of the resume CLOB in the data base.                    */
 /* Note: this program must have 'write' access to the directory   */
 /*       designated in the 'name_string' variable.                */
 name_string = '/SYSTEM/tmp/pliclob1.txt';
 hv_clob_file.sql_lob_file_name_len = length(name_string);
 hv_clob_file.sql_lob_file_name     = name_string;
 hv_clob_file.sql_lob_file_options  = ior(sql_file_overwrite);

 hv_empno = '000130';
 exec sql
    select resume into :hv_clob_file
    from dsn8a10.emp_photo_resume
    where empno = :hv_empno;
 display('file1   sqlca.sqlcode = ' || sqlca.sqlcode );

 /* Next, a CLOB locator is used to locate the resume CLOB for     */
 /* employee number '000130' in the data base.  Then a series of   */
 /* DB2 SET statements using the posstr DB2 function finds the     */
 /* beginning position of each section within the resume.          */
 exec sql
    select resume into :hv_loc_resume
    from dsn8a10.emp_photo_resume
    where empno = :hv_empno;
 display('select resume sqlcode = '|| sqlca.sqlcode);

 exec sql set :start_resume =
          (posstr(:hv_loc_resume, 'Resume:'));
 display('first set sqlcode     = '|| sqlca.sqlcode);

 exec sql set :start_pers_info =
          (posstr(:hv_loc_resume, 'Personal Information'));
 display('second set sqlcode    = '|| sqlca.sqlcode);
```

Figure 9. pliclob sample program

```
  exec sql set :start_dept_info =
           (posstr(:hv_loc_resume, 'Department Information'));
  display('third set sqlcode     = '|| sqlca.sqlcode);

  exec sql set :start_education =
           (posstr(:hv_loc_resume, 'Education'));
  display('fourth set sqlcode    = '|| sqlca.sqlcode);

  exec sql set :start_work_hist =
           (posstr(:hv_loc_resume, 'Work History'));
  display('fifth set sqlcode     = '|| sqlca.sqlcode);

  exec sql set :start_interests =
           (posstr(:hv_loc_resume, 'Interests'));
  display('sixth set sqlcode     = '|| sqlca.sqlcode);

  /* Finally, by using the CLOB locator and the start references    */
  /* of each section in the resume, along with the DB2 substr and   */
  /* concatenate (||) functions, the resume CLOB is written out to  */
  /* a second file in a slightly different format:                  */
  /*   1. the Personal Information section is omitted due to        */
  /*      privacy concerns.                                         */
  /*   2. the sections within the resume are written out in this    */
  /*      order: Resume, Work History, Education then Department    */
  /*      Information.                                              */
  /*                                                                */
  /* After the second file is written out, the changes to the       */
  /* resume CLOB can be verified by comparing the contents of the   */
  /* two files pliclob1.txt and pliclob2.txt.                       */
  /*                                                                */
  /* Note: this program must have 'write' access to the directory   */
  /*       designated in the 'name_string' variable.                */
  name_string = '/SYSTEM/tmp/pliclob2.txt';
  hv_clob_file.sql_lob_file_name_len = length(name_string);
  hv_clob_file.sql_lob_file_name     = name_string;
  hv_clob_file.sql_lob_file_options  = ior(sql_file_overwrite);

  exec sql
   values ( substr(:hv_loc_resume,:start_resume,
                   :start_pers_info-:start_resume)
         || substr(:hv_loc_resume,:start_work_hist,
                   :start_interests-:start_work_hist)
         || substr(:hv_loc_resume,:start_education,
                   :start_work_hist-:start_education)
         || substr(:hv_loc_resume,:start_dept_info,
                   :start_education-:start_dept_info)
         )
    into :hv_clob_file;

  display('file2   sqlca.sqlcode = ' || sqlca.sqlcode );
  display('End   pliclob');

 end;
```

pliclob sample program (continued)

Suppressing SQL preprocessor messages

You can use the IBM-supplied compiler user exit (IBMUEXIT) to suppress a message or to change the severity of a message.

See “Example of suppressing SQL messages” on page 490 for an example of suppressing the preprocessor messages by modifying the user exit.

CICS preprocessor

You can use EXEC CICS statements in PL/I applications that run as transactions under CICS.

If you do not specify the PP(CICS) option, EXEC CICS statements are parsed and variable references in them are validated. If they are correct, no messages are issued as long as the NOCOMPILE option is in effect. If you do not invoke the CICS translator and the COMPILE option is in effect, the compiler will issue S-level messages.

The compiler will invoke the CICS preprocessor if you specify the CICS suboption of the PP option. For compatibility, the compiler will also invoke the CICS preprocessor if any of these options is in effect: CICS, XOPT, or XOPTS. However, you should not specify any of these options together with the PP(CICS) option.

Programming and compilation considerations

When you are developing programs for execution under CICS, all the EXEC CICS commands must be translated in one of two ways:

- By the command language translator provided by CICS in a job step before the PL/I compilation
- By the PL/I CICS preprocessor as part of the PL/I compilation (this requires CICS TS 2.2 or later)

To use the CICS preprocessor, you must also specify the PP(CICS) and DFT(EBCDIC) compile time options. All data passed to CICS must be in the NATIVE format.

Unless you specify CICS as one of the suboptions of the PP(CICS) option, the compiler will flag any EXEC CICS statements in the source. Similarly, it will flag any EXEC CPSM or EXEC DLI statements if you do not specify CPSM or DLI respectively as a suboption of the PP(CICS) option.

If your CICS program is a MAIN procedure, you must also compile it with the SYSTEM(CICS) or SYSTEM(MVS) option. If you compile with SYSTEM(MVS), the PTFs for runtime APAR PQ91318 must be applied. NOEXECOPS is implied with this option and all parameters passed to the MAIN procedure must be POINTERs. For a description of the SYSTEM compile time option, see “SYSTEM” on page 87.

If your want your CICS program to be reentrant and if your program uses FILEs or CONTROLLED variables, you must compile it with the NOWRITABLE as well.

If your CICS program includes any files or uses any macros that contain EXEC CICS statements, you must also run the MACRO preprocessor before your code is translated (in either of the ways described above). If you are using the CICS preprocessor, you can specify this with one PP option as illustrated in the following example:

```
pp (macro(...)  cics(...)  )
```

The CICS preprocessor will add a set of declares for CICS variables and APIs to all non-nested procedures. Consequently, it keeps tracks of all statements that require a matching END statement, and if some of these statements are missing or incorrect, the preprocessor might be misled and not insert these declares. The preprocessor will also terminate with a severe message if such statements are nested more than 150 deep.

Finally, in order to use the CICS preprocessor, you must have the CICS SDFHLOAD dataset as part of the STEPLIB DD for the PL/I compiler.

CICS preprocessor options

There are many options supported by the CICS translator.

For a description of these options, see the *CICS Application Programming Guide*.

Note that these options should be enclosed in quotation marks (single or double, as long as they match). For instance, to invoke the CICS preprocessor with the EDF option, you must specify the option PP(CICS('EDF')).

Coding CICS statements in PL/I applications

You can code CICS statements in your PL/I applications by using the language defined in *CICS on Open Systems Application Programming Guide*. Specific requirements for your CICS code are described in the sections that follow.

Embedding CICS statements

If you use the CICS translator rather than the integrated preprocessor, the first statement of your PL/I program must be a PROCEDURE statement. You can add CICS statements to your program wherever executable statements can appear. Each CICS statement must begin with EXEC (or EXECUTE) CICS and end with a semicolon (;).

For example, the GETMAIN statement might be coded as follows:

```
EXEC CICS GETMAIN SET(BLK_PTR) LENGTH(STG(BLK));
```

Comments:

In addition to the CICS statements, PL/I comments can be included in embedded CICS statements wherever a blank is allowed.

Continuation for CICS statements:

Line continuation rules for CICS statements are the same as those for other PL/I statements.

Including code:

If included code contains EXEC CICS statements or if your program uses PL/I macros that generate EXEC CICS statements, you must use one of the following options:

- The MACRO compile-time option
- The MACRO option of the PP option (before the CICS option of the PP option)

Margins:

CICS statements must be coded within the columns specified in the MARGINS compile-time option.

Statement labels:

EXEC CICS statements, like PL/I statements, can have a label prefix.

Writing CICS transactions in PL/I

You can use PL/I with CICS facilities to write application programs (transactions) for CICS subsystems. If you do this, CICS provides facilities, which would normally be provided directly by the operating system, to the PL/I program. These facilities include most data management facilities and all job and task management facilities.

You must observe the following restrictions on PL/I CICS programs:

- Macro-level CICS is not supported.
- You cannot use PL/I input or output except for the following:
  - PUT FILE(SYSPRINT)
  - DISPLAY
  - CALL PLIDUMP
- You cannot use the PLISRTx built-in subroutines.
- Routines written in a language other than PL/I cannot be called from a PL/I CICS program if those routines contain their own EXEC CICS statements. If you want to communicate with a non-PL/I program that contains EXEC CICS statements, you must use EXEC CICS LINK or EXEC CICS XCTL.

Although PUT FILE(SYSPRINT) is permitted under CICS, you should generally not use it in production programs because it might degrade performance.

Because the CICS EIB address is only generated by either the CICS translator or the PL/I CICS preprocessor for an OPTIONS(MAIN) program, you must establish the addressability to the EIB for the OPTIONS(FETCHABLE) routine in one of the following ways:

- Use this command:

  ```
  EXEC CICS ADDRESS EIB(DFHEIPTR)
  ```

- Pass the EIB address as an argument to the CALL statement that invokes the external procedure.

Error-handling

Language Environment prohibits the use of some EXEC CICS commands in any PL/I ON-unit or in any code called from a PL/I ON-unit.

The following EXEC CICS commands are not allowed in ON-unit:

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

All other EXEC CICS commands are allowed within an ON-unit. However, you must code them by using the NOHANDLE option, the RESP option, or the RESP2 option.

Chapter 3. Using PL/I cataloged procedures

This chapter describes the standard cataloged procedures supplied by IBM for use with the IBM Enterprise PL/I for z/OS compiler. It explains how to invoke them, and how to temporarily or permanently modify them.

The Language Environment SCEERUN data set must be located in STEPLIB and accessible to the compiler when you use any of the cataloged procedures.

A cataloged procedure is a set of job control statements, stored in a library, that includes one or more EXEC statements, each of which can be followed by one or more DD statements. You can retrieve the statements by naming the cataloged procedure in the PROC parameter of an EXEC statement in the input stream.

You can use cataloged procedures to save time and reduce Job Control Language (JCL) errors. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job. You should review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for your own conventions.

IBM-supplied cataloged procedures

This section describes PL/I cataloged procedures supplied for use with Enterprise PL/I for z/OS.

For a description of the individual statements for compiling and link editing, see "Invoking the compiler under z/OS using JCL" on page 161 and the *z/OS Language Environment Programming Guide*.

The following PL/I cataloged procedures are supplied for use with Enterprise PL/I for z/OS:

IBMZC
: Compile only

IBMZCB
: Compile and bind

IBMZCBG
: Compile, bind, and run

Cataloged procedures IBMZCB and IBMZCBG use features of the program management binder introduced in DFSMS/MVS 1.4. These procedures produce a program object in a PDSE.

These cataloged procedures do not include a DD statement for the input data set; you must always provide one. The example shown in Figure 10 on page 150 illustrates the JCL statements you might use to invoke the cataloged procedure IBMZCBG to compile, bind, and run a PL/I program.

Enterprise PL/I requires a minimum REGION size of 32M. Large programs require more storage. If you do not specify REGION on the EXEC statement that invokes the cataloged procedure you are running, the compiler uses the default REGION size for your site. The default size might or might not be adequate, depending on the size of your PL/I program.

© Copyright IBM Corp. 1999, 2017

If you compile your programs with optimization turned on, the REGION size (and time) required might be much, much larger. For an example of specifying REGION on the EXEC statement, see Figure 10.

Example: Invoking a cataloged procedure

```
//COLEGO    JOB
//STEP1     EXEC IBMZCBG, REGION.PLI=32M
//PLI.SYSIN DD *
                .
                .
                .
    (insert PL/I program to be compiled here)
                .
                .
                .
/*
```

Figure 10. Invoking a cataloged procedure

Compile only (IBMZC)

The IBMZC cataloged procedure, shown in Figure 11 on page 151, includes only one procedure step, in which the options specified for the compilation are OBJECT and OPTIONS. (IBMZPLI is the symbolic name of the compiler.) In common with the other cataloged procedures that include a compilation procedure step, IBMZC does not include a DD statement for the input data set; you must always supply an appropriate statement with the qualified ddname PLI.SYSIN.

The OBJECT compile-time option causes the compiler to place the object module, in a syntax suitable for input to the linkage editor, in the standard data set defined by the DD statement with the name SYSLIN. This statement defines a temporary data set named &&LOADSET on a sequential device; if you want to retain the object module after the end of your job, you must substitute a permanent name for &&LOADSET (that is, a name that does not start with &&) and specify KEEP in the appropriate DISP parameter for the last procedure step that used the data set. You can do this by providing your own SYSLIN DD statement, as shown below. The data set name and disposition parameters on this statement will override those on the IBMZC procedure SYSLIN DD statement. In this example, the compile step is the only step in the job.

```
//PLICOMP EXEC IBMZC
//PLI.SYSLIN  DD  DSN=MYPROG,DISP=SHR
//PLI.SYSIN   DD ...
```

The term MOD in the DISP parameter in Figure 11 on page 151 allows the compiler to place more than one object module in the data set, and PASS ensures that the data set is available to a later procedure step providing a corresponding DD statement is included there.

The SYSLIN SPACE parameter allows an initial allocation of 1 cylinder and, if necessary, 15 further allocations of 1 cylinder (a total of 16 cylinders).

```
//IBMZC   PROC LNGPRFX='IBMZ.V5R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200
//*
//*****************************************************************
//*                                                               *
//* LICENSED MATERIALS - PROPERTY OF IBM                          *
//*                                                               *
//* 5655-H31  COPYRIGHT IBM CORP. 1999, 2009                      *
//* ALL RIGHTS RESERVED.                                          *
//*                                                               *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                  *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                   *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                          *
//*                                                               *
//*****************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 5 RELEASE 1 MODIFICATION
0
//*
//*  COMPILE A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V5R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*
//*  USER MUST SUPPLY //PLI.SYSIN DD STATEMENT THAT IDENTIFIES
//*   LOCATION OF COMPILER INPUT
//*
//*********************************************************************
//* COMPILE STEP
//*********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 11. Cataloged Procedure IBMZC

Compile and bind (IBMZCB)

The IBMZCB cataloged procedure, shown in Figure 12 on page 152, includes two procedure steps: PLI, which is identical to cataloged procedure IBMZC, and BIND, which invokes the Program Management binder (symbolic name IEWBLINK) to bind the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement BIND specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 8 (that is, if a severe error or an unrecoverable error occurs during compilation).

```
//IBMZCB  PROC LNGPRFX='IBMZ.V5R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200,GOPGM=GO
//*
//*******************************************************************
//*                                                                 *
//* LICENSED MATERIALS - PROPERTY OF IBM                            *
//*                                                                 *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2009                     *
//* ALL RIGHTS RESERVED.                                            *
//*                                                                 *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                    *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                     *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                            *
//*                                                                 *
//*******************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 5 RELEASE 1 MODIFICATION
0
//*
//*  COMPILE AND BIND A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V5R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*******************************************************************
//* COMPILE STEP
//*******************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*******************************************************************
//* BIND STEP
//*******************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//             PARM='XREF,COMPAT=PM3'
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
```

Figure 12. Cataloged procedure IBMZCB

The Program Management binder always places the program objects it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the program object will be placed and given the member name GO. In specifying a temporary library, the cataloged procedure assumes that you will run

the program object in the same job; if you want to retain the program object, you must substitute your own statement for the DD statement with the name SYSLMOD.

Compile, bind, and run (IBMZCBG)

The IBMZCBG cataloged procedure, shown in Figure 13 on page 154, includes three procedure steps: PLI, BIND, and GO. PLI and BIND are identical to the two procedure steps of IBMZCB, and GO runs the program object created in the step BIND. The GO step is executed only if no severe or unrecoverable errors occurred in the preceding procedure steps.

Input data for the compilation procedure step should be specified in a DD statement with the name PLI.SYSIN, and for the GO step in a DD statement with the name GO.SYSIN.

```
//IBMZCBG  PROC LNGPRFX='IBMZ.V5R1M0',LIBPRFX='CEE',
//             SYSLBLK=3200,GOPGM=GO
//*
//*****************************************************************
//*                                                               *
//* LICENSED MATERIALS - PROPERTY OF IBM                          *
//*                                                               *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2015                   *
//* ALL RIGHTS RESERVED.                                          *
//*                                                               *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                  *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                   *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                          *
//*                                                               *
//*****************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 5 RELEASE 1 MODIFICATION
0
//*
//*  COMPILE, BIND, AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V5R1M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*****************************************************************
//* COMPILE STEP
//*****************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*****************************************************************
//* BIND STEP
//*****************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//             PARM='XREF,COMPAT=PM3'
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
//*****************************************************************
//* RUN STEP
//*****************************************************************
//GO       EXEC PGM=*.BIND.SYSLMOD,COND=((8,LT,PLI),(8,LE,BIND))
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 13. Cataloged procedure IBMZCBG

Invoking a cataloged procedure

To invoke a cataloged procedure, specify its name in the PROC parameter of an EXEC statement.

For example, to use the cataloged procedure IBMZC, you can include the following statement in the appropriate position among your other job control statements in the input stream:

```
//stepname EXEC PROC=IBMZC
```

You do not need to code the keyword PROC. If the first operand in the EXEC statement does not begin with PGM= or PROC=, the job scheduler interprets it as the name of a cataloged procedure. The following statement is equivalent to that given above:

```
//stepname EXEC IBMZC
```

If you include the parameter MSGLEVEL=1 in your JOB statement, the operating system will include the original EXEC statement in its listing, and will add the statements from the cataloged procedure. In the listing, cataloged procedure statements are identified by XX or X/ as the first two characters; X/ signifies a statement that was modified for the current invocation of the cataloged procedure.

You might be required to modify the statements of a cataloged procedure for the duration of the job step in which it is invoked, either by adding DD statements or by overriding one or more parameters in the EXEC or DD statements. For example, cataloged procedures that invoke the compiler require the addition of a DD statement with the name SYSIN to define the data set containing the source statements. Also, whenever you use more than one standard link-edit procedure step in a job, you must modify all but the first cataloged procedure that you invoke if you want to run more than one of the load modules.

Specifying multiple cataloged procedure invocations

You can invoke different cataloged procedures, or invoke the same cataloged procedure several times, in the same job.

No special problems are likely to arise unless more than one of these cataloged procedures involves a link-edit procedure step, in which case you must take the following precautions to ensure that all your load modules can be run.

When the linkage editor creates a load module, it places the load module in the standard data set defined by the DD statement with the name SYSLMOD. When the binder creates a program object, it places the program object in the PDSE defined by the DD statement with the name SYSLMOD. In the absence of a linkage editor NAME statement, the linkage editor or the binder uses the member name specified in the DSNAME parameter as the name of the module. In the standard cataloged procedures, the DD statement with the name SYSLMOD always specifies a temporary library &&GOSET with the member name GO.

If you use the cataloged procedure IBMZCBG twice within the same job to compile, bind, and run two PL/I programs, and do not name each of the two program objects that the binder creates, the first program object runs twice, and the second one not at all.

To prevent this, use one of the following methods:

- Delete the library &&GOSET at the end of the GO step. In the first invocation of the cataloged procedure at the end of the GO step, add a DD statement with the syntax:

```
//GO.SYSLMOD DD DSN=&&GOSET,
//   DISP=(OLD,DELETE)
```

- Modify the DD statement with the name SYSLMOD in the second and subsequent invocations of the cataloged procedure so as to vary the names of the load modules; for example, `//BIND.SYSLMOD DD DSN=&&GOSET(GO1)`, and so on.
- Use the NAME linkage editor option to give a different name to each program object and change each job step EXEC statement to specify the running of the program object with the name for that job step.

To assign a membername to the program object, you can use the linkage editor NAME option with the DSNAME parameter on the SYSLMOD DD statement. When you use this procedure, the membername **must** be identical to the name on the NAME option if the EXEC statement that runs the program refers to the SYSLMOD DD statement for the name of the module to be run.

Another option is to give each program a different name by using GOPGM on the EXEC procedure statement, as in the following example:

```
//  EXEC IBMZCBG,GOPGM=GO2
```

Modifying the PL/I cataloged procedures

You can modify a cataloged procedure temporarily by including parameters in the EXEC statement that invokes the cataloged procedure, or by placing additional DD statements after the EXEC statement.

Temporary modifications apply only for the duration of the job step in which the procedure is invoked. They do not affect the master copy of the cataloged procedure in the procedure library.

Temporary modifications can apply to EXEC or DD statements in a cataloged procedure. To change a parameter of an EXEC statement, you must include a corresponding parameter in the EXEC statement that invokes the cataloged procedure. To change one or more parameters of a DD statement, you must include a corresponding DD statement after the EXEC statement that invokes the cataloged procedure. Although you cannot add a new EXEC statement to a cataloged procedure, you can always include additional DD statements.

EXEC statement

You can modify a cataloged procedure temporarily by including parameters in the EXEC statement that invokes the cataloged procedure.

If a parameter of an EXEC statement that invokes a cataloged procedure has an unqualified name, the parameter applies to all the EXEC statements in the cataloged procedure. The effect on the cataloged procedure depends on the parameters, as follows:

- PARM applies to the first procedure step and nullifies any other PARM parameters.
- COND and ACCT apply to all the procedure steps.

- TIME and REGION apply to all the procedure steps and override existing values.

For example, the following statement has these effects:

```
//stepname EXEC IBMZCBG,PARM='OFFSET',REGION=32M
```

- Invokes the cataloged procedure IBMZCBG.
- Substitutes the option OFFSET for OBJECT and OPTIONS in the EXEC statement for procedure step PLI.
- Nullifies the PARM parameter in the EXEC statement for procedure step BIND.
- Specifies a region size of 32M for all three procedure steps.

To change the value of a parameter in only one EXEC statement of a cataloged procedure, or to add a new parameter to one EXEC statement, you must identify the EXEC statement by qualifying the name of the parameter with the name of the procedure step. For example, to alter the region size for procedure step PLI only in the preceding example, code as follows:

```
//stepname EXEC PROC=IBMZCBG,PARM='OFFSET',REGION.PLI=90M
```

A new parameter specified in the invoking EXEC statement completely overrides the corresponding parameter in the procedure EXEC statement.

You can nullify all the options specified by a parameter by coding the keyword and equal sign without a value. For example, to suppress the bulk of the linkage editor listing when invoking the cataloged procedure IBMZCBG, code as follows:

```
//stepname EXEC IBMZCBG,PARM.BIND=
```

DD statement

You can modify a cataloged procedure temporarily by placing additional DD statements after the EXEC statement.

To add a DD statement to a cataloged procedure, or to modify one or more parameters of an existing DD statement, you must include a DD statement with the form `procstepname.ddname` in the appropriate position in the input stream. If `ddname` is the name of a DD statement already present in the procedure step identified by `procstepname`, the parameters in the new DD statement override the corresponding parameters in the existing DD statement; otherwise, the new DD statement is added to the procedure step. For example, the following statement adds a DD statement to the procedure step PLI of cataloged procedure IBMZC:

```
//PLI.SYSIN DD *
```

The following statement modifies the existing DD statement SYSPRINT (causing the compiler listing to be transmitted to the system output device of class C).

```
//PLI.SYSPRINT DD SYSOUT=C
```

Overriding DD statements must appear after the procedure invocation and in the same order as they appear in the cataloged procedure. Additional DD statements can appear after the overriding DD statements are specified for that step.

To override a parameter of a DD statement, code either a revised form of the parameter or a replacement parameter that performs a similar function (for example, SPLIT for SPACE). To nullify a parameter, code the keyword and equal sign without a value. You can override DCB subparameters by coding only those

you want to modify; that is, the DCB parameter in an overriding DD statement does not necessarily override the entire DCB parameter of the corresponding statement in the cataloged procedures.

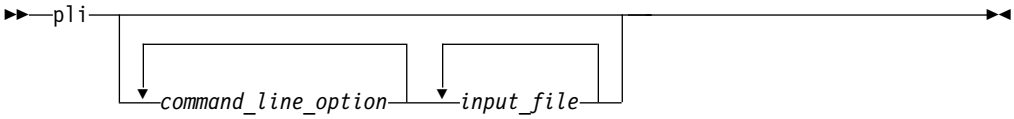
Chapter 4. Compiling your program

This chapter describes how to invoke the compiler under z/OS UNIX System Services (z/OS UNIX) and the job control statements used for compiling under z/OS.

The Language Environment SCEERUN data set must be accessible to the compiler when you compile your program.

Invoking the compiler under z/OS UNIX

To compile your program under the z/OS UNIX environment, use the **pli** command.



command_line_option
You can specify a **command_line_option** in the following ways:
- **-q***option*
- Option flag (usually a single letter preceded by -)

If you specify compile time options on the command line, the format differs from the format if you set them in your source file using %PROCESS statements. See "Specifying compile-time options under z/OS UNIX" on page 160.

input_file
The z/OS UNIX file specification for your program files.

If you omit the extension from your file specification, the compiler assumes the extension `.pli`. If you omit the complete path, the compiler assumes the current directory.

Input files

The **pli** command compiles PL/I source files, links the resulting object files with any object files and libraries specified on the command line in the order indicated, and produces a single executable file.

The **pli** command accepts the following types of files:

Source files—.pli
All `.pli` files are source files for compilation. The **pli** command sends source files to the compiler in the order they are listed. If the compiler cannot find a specified source file, it produces an error message and the **pli** command proceeds to the next file if one exists.

All HFS source files must be line-delimited and encoded in EBCDIC.

Object files—.o
All `.o` files are object files. The **pli** command sends all object files along with library files to the linkage editor at link-edit time unless you specify the **-c** option. After it compiles all the source files, the compiler invokes the linkage

© Copyright IBM Corp. 1999, 2017

editor to link-edit the resulting object files with any object files specified in the input file list, and produces a single executable output file.

Library files—.a
The **pli** command sends all of the library files (**.a** files) to the linkage editor at link-edit time.

Specifying compile-time options under z/OS UNIX

Enterprise PL/I provides compile-time options to change any of the default settings of the compiler. You can specify options on the command line, and they remain in effect for all compilation units in the file unless %PROCESS statements in your source program override them.

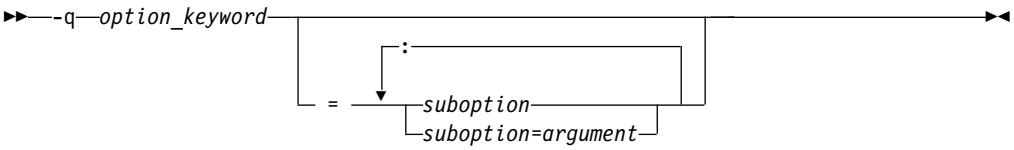
See "Compile-time option descriptions" on page 3 for a description of these options.

When you specify options on the command line, they override the default settings of the option. They are overridden by options set in the source file.

You can specify compile-time options on the command line in three ways:
- **-q***option_keyword* (compiler-specific)
- Single and multiletter flags
- -q+/u/myopts.txt

-qoption_keyword

You can specify options on the command line by using the **-q***option* format.



You can have multiple **-q***options* on the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the **-q** in lowercase.

Some compile-time options allow you to specify suboptions. These suboptions are indicated on the command line with an equal sign following the **-q***option_keyword*. Multiple suboptions must be separated with a colon(:) and no intervening blanks.

An option, for example, that contains multiple suboptions is RULES. To specify RULES(LAXDCL) on the command line, enter the following command:

```
-qrules=ibm:laxdcl
```

The LIMITS option is slightly more complex because each of its suboptions also has an argument. You can specify LIMITS(EXTNAME(31),FIXEDDEC(15)) on the command line as shown in the following example:

```
-qlimits=extname=31:fixeddec=15
```

Related information:

"RULES" on page 70
The RULES option allows or disallows certain language capabilities and lets you choose semantics when alternatives are available. It can help you diagnose common programming errors.

"LIMITS" on page 45
The LIMITS option specifies various implementation limits.

Single and multiletter flags

The z/OS UNIX family of compilers uses a number of common conventional flags. Each language has its own set of additional flags.

Some flag options have arguments that form part of the flag. In the following example, `/home/test3/include` is an include directory to be searched for INCLUDE files.

```
pli samp.pli -I/home/test3/include
```

Each flag option should be specified as a separate argument.

Table 13. Compile-time option flags supported by Enterprise PL/I under z/OS UNIX

Option	Description
`-c`	Compile only.
`-e`	Create names and entries for a fetchable load module.
`-I`<*dir*>*	Add path <*dir*> to the directories to be searched for INCLUDE files. `-I` must be followed by a path and only a single path is allowed per `-I` option. To add multiple paths, use multiple `-I` options. There should not be any spaces between `-I` and the path name.
`-0, -02`	Optimize generated code. This option is equivalent to -qOPT=2.
`-q`<*option*>*	Pass it to the compiler. <*option*> is a compile-time option. Each option should be delimited by a comma and each suboption should be delimited by an equal sign or a colon. There should not be any spaces between **-q** and <*option*>.
`-v`	Display compile and link steps, and execute them.
`-#`	Display compile and link steps, but do not execute them.

Note: *You must specify an argument where indicated; otherwise, the results are unpredictable.

Invoking the compiler under z/OS using JCL

Although you will probably use cataloged procedures rather than supply all the JCL statements required for a job step that invokes the compiler, you must be familiar with these statements so that you can make the best use of the compiler and, if necessary, override the statements of the cataloged procedures.

So-called "batch compilation", whereby one compilation produces more than one object deck, is not supported.

Invoking the compiler by BPXBATCH is also not supported.

The following section describes the JCL needed for compilation. The IBM-supplied cataloged procedures described in "IBM-supplied cataloged procedures" on page 149 contain these statements. You need to code them yourself only if you are not using the cataloged procedures.

EXEC statement

The basic EXEC statement is `//stepname EXEC PGM`.

512K is required for the REGION parameter of this statement.

If you compile your programs with optimization turned on, the REGION size (and time) required might be much, much larger.

The PARM parameter of the EXEC statement can be used to specify one or more of the optional facilities provided by the compiler. These facilities are described under "Specifying options in the EXEC statement" on page 165. See Chapter 1, "Using compiler options and facilities," on page 3 for a description of the options.

DD statements for the standard data sets

The compiler requires several standard data sets. The number of data sets depends on the optional facilities specified. You must define these data sets in DD statements.

You must define these data sets in DD statements with the standard ddnames shown, together with other characteristics of the data sets, in Table 14. The DD statements SYSIN, SYSUT1, and SYSPRINT are always required.

You can store any of the standard data sets on a direct access device, but you must include the SPACE parameter in the DD statement. This parameter defines the data set to specify the amount of auxiliary storage required. The amount of auxiliary storage allocated in the IBM-supplied cataloged procedures should suffice for most applications.

Table 14. Compiler standard data sets

Standard DDNAME	Contents of data set	Possible device classes[1]	Record format (RECFM)	Record size (LRECL)
SYSDEBUG	TEST(SEPARATE) output	SYSDA	F,FB	>=80 and <=1024
SYSDEFSD	XINFO(DEF) output	SYSDA	F,FB	128
SYSIN	Input to the compiler	SYSSQ	F,FB,U	<101(100)
			VB,V	<105(104)
SYSLIB	Source statements for INCLUDE files	SYSDA	F,FB,U	<101
			V,VB	<105
SYSLIN	Object module	SYSSQ	FB	80
SYSPRINT	Listing, including messages	SYSSQ	VBA	137
SYSPUNCH	Preprocessor output, compiler output	SYSSQ SYSCP	FB	80 or MARGINS() value
SYSUT1	Temporary workfile	SYSDA	F	4051
SYSUT2	Temporary workfile	SYSDA	FB	3200
SYSUT3	Temporary workfile	SYSDA	FB	3200
SYSXMLSD	XINFO(XML) output	SYSDA	VB	16383
SYSADATA	XINFO(MSG) output	SYSDA	U	1024

Table 14. Compiler standard data sets (continued)

Standard DDNAME	Contents of data set	Possible device classes[1]	Record format (RECFM)	Record size (LRECL)

1. **Descriptions of device classes**
   SYSSQ
   Sequential device
   SYSDA
   Direct access device
   Block size can be specified except for SYSUT1. The block size and logical record length for SYSUT1 is chosen by the compiler.

Notes:

1. The only value for compile-time SYSPRINT that can be overridden is BLKSIZE.
2. SYSUT2 and SYSUT3 are required only under LP(64) and only if the GONUMBER or TEST option is in effect.

Input (SYSIN)

Input to the compiler must be a data set defined by a DD statement with the name SYSIN.

This data set must have the CONSECUTIVE organization. The input must be one or more external PL/I procedures. If you want to compile more than one external procedure in a single job or job step, precede each procedure, except possibly the first, with a %PROCESS statement.

80-byte records are commonly used as the input medium for PL/I source programs. The input data set can be on a direct access device or on some other sequential media. The input data set can contain either fixed-length records (blocked or unblocked), variable-length records (coded or uncoded), or undefined-length records. The maximum record size is 100 bytes.

The maximum number of lines in the input file is 999999.

When data sets are concatenated for input to the compiler, the concatenated data sets must have similar characteristics (for example, block size and record format).

Output (SYSLIN, SYSPUNCH)

Output in the form of one or more object modules from the compiler will be stored in the data set SYSLIN if you specify the OBJECT compile-time option. This data set is defined by the DD statement.

The object module is always in the form of 80-byte fixed-length records, blocked or unblocked. If the BLKSIZE is specified for SYSLIN and is not 80, the LRECL must be specified as 80.

The SYSLIN DD must name either a temporary data set or a permanent data set: it cannot specify a concatenation of data sets of any type.

The SYSLIN DD must specify a sequential dataset, not a PDS or PDSE.

The data set defined by the DD statement with the name SYSPUNCH is also used to store the output from the preprocessor if you specify the MDECK compile-time option.

Temporary workfile (SYSUT1)

The compiler requires a data set for use as a temporary workfile. It is defined by a DD statement with the name SYSUT1, and is known as the *spill file*. It must be on a direct access device, and must not be allocated as a multivolume data set.

The spill file is used as a logical extension to main storage and is used by the compiler and by the preprocessor to contain text and dictionary information. The LRECL and BLKSIZE for SYSUT1 is chosen by the compiler based on the amount of storage available for spill file pages.

The DD statements given in this publication and in the cataloged procedures for SYSUT1 request a space allocation in blocks of 1024 bytes. This is to ensure that adequate secondary allocations of direct access storage space are acquired.

Temporary workfile (SYSUT2, SYSUT3)

The compiler requires SYSUT2 and SYSUT3 as temporary data sets when compiling programs under the LP(64) option and the GONUMBER or TEST option is in effect.

For large programs, if there is not enough space available in the SYSUT2 or the SYSUT3 data set, then the compiler might abend.

Listing (SYSPRINT)

The compiler generates a listing that includes all the source statements that it processed, information relating to the object module, and, when necessary, messages.

Most of the information included in the listing is optional, and you can specify those parts that you require by including the appropriate compile-time options. For details about the information that can appear and the associated compile-time options, see "Using the compiler listing" on page 102.

You must define the data set, in which you want the compiler to store its listing, in a DD statement with the name SYSPRINT. This data set must have the CONSECUTIVE organization. Although the listing is usually printed, it can be stored on any sequential or direct access device. For printed output, the following statement will suffice if your installation follows the convention that output class A refers to a printer:

```
//SYSPRINT DD SYSOUT=A
```

Source Statement Library (SYSLIB)

If you use the %INCLUDE statement to introduce source statements into the PL/I program from a library, you can either define the library in a DD statement with the name SYSLIB, or choose your own ddname (or ddnames) and specify a ddname in each %INCLUDE statement.

The DD statement should specify a PDS or PDSE, but not the actual member. For example, to include the file HEADER from the library SYSLIB by using the data set INCLUDE.PLI, you can use one of the following %INCLUDE statements:

- `%INCLUDE HEADER;`
- `%INCLUDE SYSLIB( HEADER );`

The DD statement should be specified as follows:

```
SYSLIB  DD DISP=SHR,DSN=INCLUDE.PLI
```

But the following statement is not valid:

```
SYSLIB  DD DISP=SHR,DSN=INCLUDE.PLI(HEADER)
```

All %INCLUDE files must have the same record format (fixed, variable, or undefined), the same logical record length, and the same left and right margins as the SYSIN source file.

The BLOCKSIZE of the library must be less than or equal to 32760 bytes.

The maximum number of lines in any one include file is 999999.

Specifying options

For each compilation, the IBM-supplied or installation default for a compile-time option applies unless it is overridden by specifying the option in a %PROCESS statement or in the PARM parameter of an EXEC statement.

An option specified in the PARM parameter overrides the default value, and an option specified in a %PROCESS statement overrides both the value specified in the PARM parameter and the default value.

Note: When conflicting attributes are specified either explicitly or implicitly by the specification of other options, the latest implied or explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden in this way.

Specifying options in the EXEC statement

To specify options in the EXEC statement, code `PARM=` followed by the list of options. You can list the options in any order. You must separate the options with commas, and enclose the list within single quotation marks.

See the following example:

```
//STEP1 EXEC PGM=IBMZPLI,PARM='OBJECT,LIST'
```

If any option has quotation marks, for example `MARGINI('c')`, you must duplicate the quotation marks. The length of the option list must not exceed 100 characters, including the separating commas. However, you can use the abbreviated syntax of options, if available, to save space. If you need to continue the statement onto another line, you must enclose the list of options in parentheses (instead of in quotation marks), enclose the options list on each line in quotation marks, and ensure that the last comma on each line except the last line is outside the quotation marks. The following example illustrates all these points:

```
//STEP1 EXEC PGM=IBMZPLI,PARM=('AG,A',
//       'C,F(I)',
// 'M,MI(''X''),NEST,STG,X')
```

If you are using a cataloged procedure and want to specify options explicitly, you must include the PARM parameter in the EXEC statement that invokes it, qualifying the keyword PARM with the name of the procedure step that invokes the compiler, as in the following example:

```
//STEP1 EXEC nnnnnnn,PARM.PLI='A,LIST'
```

Specifying options in the EXEC statement using an options file

Another way to specify options in the EXEC statement is by declaring all your options in an options file and coding the following:

```
//STEP1 EXEC PGM=IBMZPLI,PARM='+DD:OPTIONS'
```

This method allows you to provide a consistent set of options that you frequently use. This is especially effective if you want other programmers to use a common set of options. It also gets you past the 100-character limit.

The MARGINS option does not apply to options files: the data in column 1 will be read as part of the options. Also, if the file is F-format, any data after column 72 will be ignored.

The parm string can contain "normal" options and can point to more than one options file. For instance, to specify the option LIST as well as options from both the file in the GROUP DD and the file in the PROJECT DD, you can specify the following:

```
PARM='LIST +DD:GROUP +DD:PROJECT'
```

The options in the PROJECT file have precedence over options in the GROUP file.

Also, in this case, the LIST option might be turned off by a NOLIST option specified in either of the options files. To ensure that the LIST option is on, you can specify the following:

```
PARM='+DD:GROUP +DD:PROJECT LIST'
```

You can also use options files under z/OS UNIX. For example, in z/OS UNIX, to compile sample.pli with options from the file /u/pli/group.opt, you can use the following command:

```
pli -q+/u/pli/group.opt  sample.pli
```

Earlier releases of the compiler used the character '@' as the trigger character that preceded the options file specification. This character is not part of the invariant set of EBCDIC code points, and for that reason the character '+', which is invariant, is preferred. However, the '@' character can still be used as long as it is specified with the hex value '7C'x.

Chapter 5. Link-editing and running for 31-bit programs

After compilation with LP(32), your 31-bit program consists of one or more object modules that contain unresolved references to each other, as well as references to the Language Environment runtime library. These references are resolved during link-editing (statically) or during execution (dynamically).

After you compile your PL/I program, the next step is to link and run your program with test data to verify that it produces the results you expect.

Language Environment provides the runtime environment and services you need to execute your program. For instructions on linking and running PL/I and all other Language Environment-conforming language programs, see the *z/OS Language Environment Programming Guide*. For information about migrating your existing PL/I programs to Language Environment, see the *Enterprise PL/I for z/OS Compiler and Run-Time Migration Guide*.

Link-edit considerations for 31-bit programs

If you compile with the option RENT or the option LIMITS(EXTNAME(*n*)) with *n* > 8, you must use a PDSE for your linker output.

Using the binder in 31-bit programs

You must place the binder output into a PDSE.

When linking a DLL, you must specify any needed definition side-decks during the bind step.

Using the ENTRY card

If you are building a module that will be fetched and that has an Enterprise PL/I routine as its entry point, the ENTRY card should specify the name of that PL/I entry point.

If the module is to be fetched from Enterprise PL/I, you can specify CEESTART on the ENTRY card, although this is strongly not recommended. However, if the module is to be fetched from COBOL or the assembler, the ENTRY card absolutely must specify the name of the PL/I entry point into the module and not CEESTART.

Runtime considerations for 31-bit programs

You can specify runtime options as parameters passed to the program initialization routine. You can also specify runtime options in the PLIXOPT variable. It might also prove beneficial, from a performance standpoint, if you alter your existing programs by using the PLIXOPT variable to specify your runtime options and recompiling your programs.

For a description of using PLIXOPT, see the *z/OS Language Environment Programming Guide*.

To simplify input/output at the terminal, various conventions have been adopted for stream files that are assigned to the terminal. Three areas are affected:

© Copyright IBM Corp. 1999, 2017

1. Formatting of PRINT files
2. The automatic prompting feature
3. Spacing and punctuation rules for input

Note: No prompting or other facilities are provided for record I/O at the terminal, so you are strongly advised to use stream I/O for any transmission to or from a terminal.

Formatting conventions for PRINT files

When a PRINT file is assigned to the terminal, it is assumed that it will be read as it is being printed. Spacing is therefore reduced to a minimum to reduce printing time.

The following rules apply to the PAGE, SKIP, and ENDPAGE keywords:

- PAGE options or format items result in three lines being skipped.
- SKIP options or format items larger than SKIP (2) result in three lines being skipped. SKIP (2) or less is treated in the usual manner.
- The ENDPAGE condition is never raised.

Changing the format on PRINT files for 31-bit programs

If you want normal spacing to apply to output from a PRINT file at the terminal, you must supply your own tab table for PL/I.

Follow these steps:

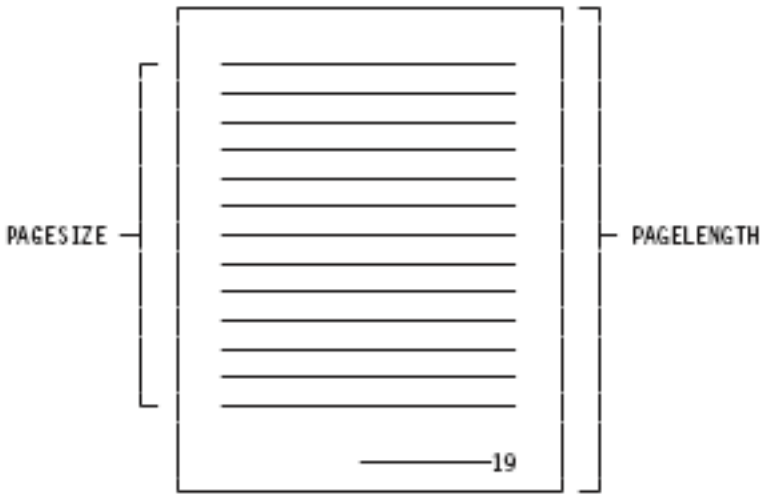
1. Declare an external structure called PLITABS in the main program or in a program linked with the main program.
2. Initializing the element PAGELENGTH to the number of lines that can fit on your page. This value differs from PAGESIZE, which defines the number of lines you want to print on the page before ENDPAGE is raised (see Figure 15 on page 169).

If you require a PAGELENGTH of 64 lines, declare PLITABS as shown in Figure 14 on page 169. For information about overriding the tab table, see "Overriding the tab control table" on page 257.

If your code contains a declare for PLITABS, ensure that the values and the first field in the PLITABS structure must are all valid. This field is supposed to hold the offset to the field specifying the number of tabs set by the structure, and the Enterprise PL/I library code will not work correctly if this is not true.

```
DCL 1 PLITABS STATIC EXTERNAL,
  ( 2   OFFSET INIT (14),
    2    PAGESIZE INIT (60),
    2    LINESIZE INIT (120),
    2    PAGELENGTH INIT (64),
    2    FILL1 INIT (0),
    2    FILL2 INIT (0),
    2    FILL3 INIT (0),
    2    NUMBER_OF_TABS INIT (5),
    2    TAB1 INIT (25),
    2    TAB2 INIT (49),
    2    TAB3 INIT (73),
    2    TAB4 INIT (97),
    2    TAB5 INIT (121)) FIXED BIN (15,0);
```

Figure 14. Declaration of PLITABS. This declaration gives the standard page size, line size, and tabulating positions.



PAGELENGTH: the number of lines that can be printed on a page

PAGESIZE: the number of lines that will be printed on a page before the ENDPAGE condition is raised

Figure 15. PAGELENGTH and PAGESIZE. PAGELENGTH defines the size of your paper; PAGESIZE defines the number of lines in the main printing area.

Automatic prompting

When the program requires input from a file that is associated with a terminal, it issues a prompt. This takes the form of printing a colon on the next line and then skipping to column 1 on the line following the colon. This gives you a full line to enter your input.

See the following example:

```
:
(space for entry of your data)
```

This type of prompt is referred to as a primary prompt.

Overriding automatic prompting

You can override the primary prompt by making a colon the last item in the request for the data. You cannot override the secondary prompt.

For example, the following two PL/I statements result in the terminal displaying output shown in Figure 16.

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));
```

```
ENTER TIME OF PERIHELION
:         (automatic prompt)
(space for entry of data)
```

Figure 16. Output with automatic prompt

However, if the first statement has a colon at the end of the output as follows, the automatic prompt is overridden; Figure 17 shows the sequence that is displayed on the terminal.

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

Figure 17. Output with no automatic prompt

Note: The override remains in force for only one prompt. You will be automatically prompted for the next item unless the automatic prompt is again overridden.

Punctuating long input lines

To transmit data that requires two or more lines of space at the terminal as one data item, you must use an SBCS hyphen as the line continuation character. Type an SBCS hyphen as the last character in each line except the last line.

For example, you must enter data as follows to transmit this sentence: `this data must be transmitted as one unit.`

```
:'this data must be transmitted -
+:as one unit.'
```

Transmission does not occur until you press ENTER after `unit.'`. The hyphen is removed. The item transmitted is called a *logical line*.

Note: To transmit a line whose last data character is a hyphen or a PL/I minus sign, enter two hyphens at the end of the line, followed by a null line as the next line. See the following example:

```
xyz--
(press ENTER only on this line)
```

Punctuating GET LIST and GET DATA statements

For GET LIST and GET DATA statements, a comma is added to the end of each logical line transmitted from the terminal, if the programmer omits it. Thus there is no need to enter blanks or commas to delimit items if they are entered on separate logical lines.

Given the PL/I statement `GET LIST(A,B,C);`, you can enter data as follows at the terminal:

```
:1
+:2
+:3
```

This rule also applies when you enter character-string data. Therefore, a character string must transmit as one logical line. Otherwise, commas are placed at the break points. For example, suppose you enter the following data:

```
:'COMMAS SHOULD NOT BREAK
+:UP A CLAUSE.'
```

The resulting string is `COMMAS SHOULD NOT BREAK, UP A CLAUSE.`

The comma is not added if a hyphen was used as a line continuation character.

Automatic padding for GET EDIT

For a GET EDIT statement, you do not need to enter blanks at the end of the line. The data will be padded to the specified length.

For example, given the PL/I statement `GET EDIT (NAME) (A(15));`, you can enter the 5 characters SMITH, and the data will be padded with ten blanks so that the program receives the fifteen characters:

```
'SMITH          '
```

Note: A single data item must transmit as a logical line. Otherwise, the first line transmitted will be padded with the necessary blanks and taken as the complete data item.

Use of SKIP for terminal input

All uses of SKIP for input are interpreted as SKIP(1) when the file is allocated to the terminal. SKIP(1) is treated as an instruction to ignore all unused data on the currently available logical line.

ENDFILE

You can enter end-of-file at the terminal by typing a logical line that consists of the two characters `/*`.

Any further attempts to use the file without closing it result in the ENDFILE condition being raised.

SYSPRINT considerations for 31-bit programs

The PL/I standard SYSPRINT file is shared by multiple enclaves within an application. You can issue I/O requests, for example STREAM PUT, from the same or different enclaves. These requests are handled using the standard PL/I SYSPRINT file as a file that is common to the entire application. The SYSPRINT file is implicitly closed only when the application terminates, not at the termination of the enclave.

The standard PL/I SYSPRINT file contains user-initiated output only, such as STREAM PUTs. Runtime library messages and other similar diagnostic output are directed to the Language Environment MSGFILE. See the *z/OS Language Environment Programming Guide* for details on redirecting SYSPRINT file output to the Language Environment MSGFILE.

To be shared by multiple enclaves within an application, the PL/I SYSPRINT file must be declared as an EXTERNAL FILE constant with a file name of SYSPRINT and also have the attributes STREAM and OUTPUT as well as the (implied) attribute of PRINT, when OPENed. This is the standard SYSPRINT file as defaulted by the compiler.

There exists only one standard PL/I SYSPRINT FILE within an application and this file is shared by all enclaves within the application. For example, the SYSPRINT file can be shared by multiple nested enclaves within an application or by a series of enclaves that are created and terminated within an application by the Language Environment preinitialization function. To be shared by an enclave within an application, the PL/I SYSPRINT file must be declared in that enclave. The standard SYSPRINT file cannot be shared by passing it as a file argument between enclaves. The declared attributes of the standard SYSPRINT file should be the same throughout the application, as with any EXTERNALly declared constant. PL/I does not enforce this rule. Both the TITLE option and the MSGFILE(SYSPRINT) option attempt to route SYSPRINT to another data set. As such, if the two options are used together, there will be a conflict and the TITLE option will be ignored.

Having a common SYSPRINT file within an application can be an advantage to applications that utilize enclaves that are closely tied together. However, since all enclaves in an application write to the same shared data set, this might require some coordination among the enclaves.

The SYSPRINT file is opened (implicitly or explicitly) when first referenced within an enclave of the application. When the SYSPRINT file is CLOSEd, the file resources are released (as though the file had never been opened) and all enclaves are updated to reflect the closed status.

If SYSPRINT is utilized in a multiple enclave application, the LINENO built-in function only returns the current line number until after the first PUT or OPEN in an enclave has been issued. This is required in order to maintain full compatibility with old programs.

The COUNT built-in function is maintained at an enclave level. It always returns a value of zero until the first PUT in the enclave is issued. If a nested child enclave is invoked from a parent enclave, the value of the COUNT built-in function is undefined when the parent enclave regains control from the child enclave.

The TITLE option can be used to associate the standard SYSPRINT file with different operating system data sets, keeping in mind that a particular open association has to be closed before another one is opened. This association is retained across enclaves for the duration of the open.

When opened, the TITLE option can be used to associate the standard SYSPRINT file with different operating system data sets. This association is retained across enclaves for the duration of the open.

PL/I condition handling associated with the standard PL/I SYSPRINT file retains its current semantics and scope. For example, an ENDPAGE condition raised within a child enclave will only invoke an established ON-unit within that child enclave. It does not cause invocation of an ON-unit within the parent enclave.

The tabs for the standard PL/I SYSPRINT file can vary when PUTs are done from different enclaves, if the enclaves contain a user PLITABS table.

If the PL/I SYSPRINT file is utilized as a RECORD file or as a STREAM INPUT file, PL/I supports it at an individual enclave or task level, but not as a shareable file among enclaves. If the PL/I SYSPRINT file is open at the same time with different file attributes (for example, RECORD and STREAM) in different enclaves of the same application, results are unpredictable.

SYSPRINT can also be shared between code compiled by Enterprise PL/I and by older PL/I compilers, but the following conditions must all apply:

- SYSPRINT must be declared as STREAM OUTPUT.
- The application must not be running under TSO.
- If the runtime option MSGFILE(SYSPRINT) is in effect, there must be no preinitialized programs and no stored procedures in the application.

Using MSGFILE(SYSPRINT)

Any file attributes that are specified in the ENVIRONMENT option of the file declaration for SYSPRINT STREAM PRINT are ignored.

Any attributes that are specified on the OPEN statement for SYSPRINT are ignored.

When you use the OPEN statement to open the `PL/I SYSPRINT STREAM PRINT` file, the file is marked as `opened` in the PL/I control blocks, but it is actually opened by the Language Environment.

When you use the CLOSE statements to close the `PL/I SYSPRINT STREAM PRINT` file, the file is marked as `closed` in the PL/I control blocks, but Language Environment still keeps it open.

The synchronization between the Language Environment messages and PL/I user-specified output is not provided, so the order of the output is unpredictable.

The use of MSGFILE(SYSPRINT) restricts the line size specified by the LINESIZE option to a maximum of 225 characters.

Using FETCH in your routines in 31-bit applications

In Enterprise PL/I, you can fetch routines compiled by PL/I, C, COBOL, or the assembler.

Fetching Enterprise PL/I routines in 31-bit applications

Almost all the restrictions imposed by the older PL/I compilers on fetched modules have been removed. So a fetched module can now perform the following operations:

- Fetch other modules.
- Perform any I/O operations on any PL/I file. The file can be opened either by the fetched module, by the main module, or by some other fetched module.
- ALLOCATE and FREE its own CONTROLLED variables.

There are, however, a few restrictions on an Enterprise PL/I module that is to be fetched:

1. OPTIONS(FETCHABLE) should be specified on the PROCEDURE statement of the fetched routine if there is no ENTRY card provided during the link-edit step.
2. The ENTRY card should specify the name of that PL/I entry point.
   - If the module is to be fetched from Enterprise PL/I, you can specify CEESTART on the ENTRY card, although this is strongly not recommended.
   - However, if the module is to be fetched from COBOL or the assembler, the ENTRY card absolutely must specify the name of the PL/I entry point into the module and not CEESTART.
3. If the RENT compiler option was used to compile any of the fetched code, the module must be linked as a DLL.
4. If the NORENT compiler option was used to compile the fetching code, the fetched module must satisfy at least one of the following conditions:
   - It is a MAIN module.
   - It consists only of NORENT code.
   - It has as its entry point code compiled with the C or Enterprise PL/I compiler with the NORENT option in effect. In this case, the module can also contain code compiled with the RENT option, but calling that code is not supported.
5. If the RENT compiler option was used to compile the fetching code, the ENTRY that is fetched must not be declared in the fetching module as OPTIONS(COBOL) or OPTIONS(ASM). If you want to avoid passing descriptors in this situation, you should specify the OPTIONS(NODESCRIPTOR) attribute on the ENTRY declare.
6. An Enterprise PL/I routine can not fetch itself.

   In the case of a nonreentrant and nonreusable module that is loaded multiple times, the order of processing occurs in a last-in first-out order.

   For example, if Program A loads module LOADMODA, then calls Program B, which also loads LOADMODA, and then issues a DELETE against LOADMODA, the copy of LOADMODA to be deleted is the one associated with Program B. At this point, the copy of LOADMODA associated with Program A still exists.

   In other words, the DELETE requested against LOADMODA will release the last copy that was loaded, regardless of which program issues the request.

NORENT WRITABLE code is serially usable, and for that reason, the pointer that is used to represent a FETCHABLE constant is zeroed out in the prologue code of any NORENT WRITABLE routine. While this ensures that the code is serially reusable while also providing the correct PL/I semantics, it does impose a restriction on the use of FETCH with TITLE in NORENT WRITABLE code: if a

routine that did a FETCH A TITLE('B') is exited and reentered, it must re-execute the FETCH A TITLE('B'), before executing any CALL A statements (otherwise, it would do an implicit FETCH of A (but without any TITLE) before making the CALL).

As an illustration of these restrictions, consider the compiler user exit. If you specify the EXIT compile-time option, the compiler will fetch and call a Enterprise PL/I module named IBMUEXIT.

First note that the compiler user exit must be compiled with the RENT option because the compiler expects it to be a DLL.

In accordance with Item 1 above, the PROCEDURE statement for this routine looks like:

```
ibmuexit:
  proc ( addr_Userexit_Interface_Block,
         addr_Request_Area )
  options( fetchable  );

  dcl addr_Userexit_Interface_Block  pointer byvalue;

  dcl addr_Request_Area              pointer byvalue;
```

In accordance with Item 3 above, the linker option DYNAM=DLL must be specified when linking the user exit into a DLL. The DLL must be linked either into a PDSE or into a temporary dataset (in which case DSNTYPE=LIBRARY must be specified on the SYSLMOD DD statement).

All the JCL to compile, link, and invoke the user exit is given in the JCL below in Figure 18 on page 176. The one significant difference between the sample below and the code excerpts above is that, in the code below, the fetched user exit does not receive two BYVALUE pointers to structures, but instead it receives the two structures by reference (BYADDR). In order to make this change work, the code specifies OPTIONS(NODESCRIPTOR) on each of its PROCEDURE statements.

```
//*
//*********************************************************************
//* compile the user exit
//*********************************************************************
//PLIEXIT EXEC PGM=IBMZPLI,
//STEPLIB   DD DSN=IBMZ.V5R1M0.SIBMZCMP,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*Process or('|') not('!');
*Process limits(extname(31));
*Process RENT;

 /*********************************************************************/
 /*                                                                   */
 /*  NAME - IBMUEXIT.PLI                                              */
 /*                                                                   */
 /*  DESCRIPTION                                                      */
 /*    User-exit sample program.                                      */
 /*                                                                   */
 /*    Licensed Materials - Property of IBM                           */
 /*    5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,2015.          */
 /*    All Rights Reserved.                                           */
 /*    US Government Users Restricted Rights-- Use, duplication or    */
 /*    disclosure restricted by GSA ADP Schedule Contract with        */
 /*    IBM Corp.                                                      */
 /*                                                                   */
 /*  DISCLAIMER OF WARRANTIES                                         */
 /*    The following "enclosed" code is sample code created by IBM    */
 /*    Corporation. This sample code is not part of any standard      */
 /*    IBM product and is provided to you solely for the purpose of   */
 /*    assisting you in the development of your applications.  The    */
 /*    code is provided "AS IS", without warranty of any kind.        */
 /*    IBM shall not be liable for any damages arising out of your    */
 /*    use of the sample code, even if IBM has been advised of the    */
 /*    possibility of such damages.                                   */
 /*                                                                   */
 /*********************************************************************/

 /*********************************************************************/
 /*                                                                   */
 /*  During initialization, IBMUEXIT is called.  It reads             */
 /*  information about the messages being screened from a text        */
 /*  file and stores the information in a hash table.  IBMUEXIT       */
 /*  also sets up the entry points for the message filter service     */
 /*  and termination service.                                         */
 /*                                                                   */
 /*  For each message generated by the compiler, the compiler         */
 /*  calls the message filter registered by IBMUEXIT.  The filter     */
 /*  looks the message up in the hash table previously created.       */
 /*                                                                   */
 /*  The termination service is called at the end of the compile      */
 /*  but does nothing.  It could be enhanced to generate reports     */
 /*  or do other cleanup work.                                        */
 /*                                                                   */
 /*********************************************************************/
```

Figure 18. Sample JCL to compile, link, and invoke the user exit

```
pack: package exports(*);

  Dcl
    1 Uex_UIB           native Based( null() ),
      2 Uex_UIB_Length         fixed bin(31),

      2 Uex_UIB_Exit_token     pointer,        /* for user exit's use*/

      2 Uex_UIB_User_char_str  pointer,        /* to exit option str */
      2 Uex_UIB_User_char_len  fixed bin(31),

      2 Uex_UIB_Filename_str   pointer,        /* to source filename */
      2 Uex_UIB_Filename_len   fixed bin(31),

      2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs  */
      2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs  */

      2 Uex_UIB_Exit_Routs,                    /* exit entries setat
                                                  initialization     */
        3 ( Uex_UIB_Termination,
            Uex_UIB_Message_Filter,            /* call for each msg  */
            *, *, *, * )
          limited entry (
            *,                                 /* to Uex_UIB         */
            *                                  /* to a request area  */
           );

  /*********************************************************************/
  /*                                                                   */
  /*   Request Area for Initialization exit                            */
  /*                                                                   */
  /*********************************************************************/

  Dcl 1 Uex_ISA native based( null() ),
        2 Uex_ISA_Length fixed bin(31);

  /*********************************************************************/
  /*                                                                   */
  /*   Request Area for Message_Filter exit                            */
  /*                                                                   */
  /*********************************************************************/

  Dcl 1 Uex_MFA native based( null() ),
        2 Uex_MFA_Length   fixed bin(31),
        2 Uex_MFA_Facility_Id  char(3),
        2 *                    char(1),
        2 Uex_MFA_Message_no   fixed bin(31),
        2 Uex_MFA_Severity     fixed bin(15),
        2 Uex_MFA_New_Severity fixed bin(15);  /* set by exit proc   */

  /*********************************************************************/
  /*                                                                   */
  /*   Request Area for Terminate exit                                 */
  /*                                                                   */
  /*********************************************************************/

  Dcl 1 Uex_TSA native based( null() ),
        2 Uex_TSA_Length fixed bin(31);
```

Sample JCL to compile, link, and invoke the user exit (continued)

```
/*****************************************************************/
/*                                                               */
/*   Severity Codes                                              */
/*                                                               */
/*****************************************************************/

dcl uex_Severity_Normal             fixed bin(15) value(0);
dcl uex_Severity_Warning            fixed bin(15) value(4);
dcl uex_Severity_Error              fixed bin(15) value(8);
dcl uex_Severity_Severe             fixed bin(15) value(12);
dcl uex_Severity_Unrecoverable      fixed bin(15) value(16);

/*****************************************************************/
/*                                                               */
/*   Return Codes                                                */
/*                                                               */
/*****************************************************************/

dcl uex_Return_Normal               fixed bin(15) value(0);
dcl uex_Return_Warning              fixed bin(15) value(4);
dcl uex_Return_Error                fixed bin(15) value(8);
dcl uex_Return_Severe               fixed bin(15) value(12);
dcl uex_Return_Unrecoverable        fixed bin(15) value(16);

/*****************************************************************/
/*                                                               */
/*   Reason Codes                                                */
/*                                                               */
/*****************************************************************/

dcl uex_Reason_Output               fixed bin(15) value(0);
dcl uex_Reason_Suppress             fixed bin(15) value(1);

dcl hashsize fixed bin(15) value(97);
dcl hashtable(0:hashsize-1) ptr init((hashsize) null());

dcl 1 message_item native based,
      2 message_Info,
        3 facid  char(3),
        3 msgno  fixed bin(31),
        3 newsev fixed bin(15),
        3 reason fixed bin(31),
      2 link pointer;
```

Sample JCL to compile, link, and invoke the user exit (continued)

```
ibmuexit: proc ( ue, ia  )
  options( fetchable nodescriptor );

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 ia like uex_Isa byaddr;

  dcl sysuexit     file stream input env(recsize(80));
  dcl p            pointer;
  dcl bucket       fixed bin(31);
  dcl based_Chars  char(8) based;
  dcl title_Str    char(8) var;

  ue.uex_Uib_Message_Filter = message_Filter;
  ue.uex_Uib_Termination = exitterm;


  on undefinedfile(sysuexit)
  begin;
    put edit ('** User exit unable to open exit file ')
             (A) skip;
    put skip;
    signal error;
  end;

  if ue.uex_Uib_User_Char_Len = 0 then
    do;
      open file(sysuexit);
    end;
  else
    do;
      title_Str
       = substr( ue.uex_Uib_User_Char_Str->based_Chars,
                 1, ue.uex_Uib_User_Char_Len );
      open file(sysuexit) title(title_Str);
    end;

  on error, endfile(sysuexit)
    goto done;

  allocate message_item set(p);

  /*************************************************************/
  /*                                                           */
  /*  Skip header lines and read first data line               */
  /*                                                           */
  /*************************************************************/

  get file(sysuexit) list(p->message_info) skip(3);
```

Sample JCL to compile, link, and invoke the user exit (continued)

```
      do loop;

        /*************************************************************/
        /*                                                           */
        /*  Put message information in hash table                    */
        /*                                                           */
        /*************************************************************/

        bucket = mod(p->msgno, hashsize);
        p->link = hashtable(bucket);
        hashtable(bucket) = p;

        /*************************************************************/
        /*                                                           */
        /*  Read next data line                                      */
        /*                                                           */
        /*************************************************************/

        allocate message_item set(p);
        get file(sysuexit) skip;
        get file(sysuexit) list(p->message_info);

      end;


      /*****************************************************************/
      /*                                                               */
      /*  Clean up                                                     */
      /*                                                               */
      /*****************************************************************/

      done:

      free p->message_Item;
      close file(sysuexit);

    end;

    message_Filter:
      proc ( ue, mf  )
      options( nodescriptor );

      dcl 1 ue like uex_Uib byaddr;
      dcl 1 mf like uex_Mfa byaddr;

      dcl p pointer;
      dcl bucket fixed bin(15);

      on error snap system;

      ue.uex_Uib_Reason_Code = uex_Reason_Output;
      ue.uex_Uib_Return_Code = 0;

      mf.uex_Mfa_New_Severity = mf.uex_Mfa_Severity;


      /*************************************************************/
      /*                                                           */
      /*  Calculate bucket for error message                       */
      /*                                                           */
      /*************************************************************/

      bucket = mod(mf.uex_Mfa_Message_No, hashsize);
```

Sample JCL to compile, link, and invoke the user exit (continued)

```
    /*************************************************************/
    /*                                                           */
    /*  Search bucket for error message                          */
    /*                                                           */
    /*************************************************************/

    do p = hashtable(bucket) repeat (p->link) while(p!=null())
      until (p->msgno = mf.uex_Mfa_Message_No &
             p->facid = mf.Uex_Mfa_Facility_Id);
    end;


    if p = null() then;
    else
      do;

        /*********************************************************/
        /*                                                       */
        /*  Filter error based on information in has table       */
        /*                                                       */
        /*********************************************************/

        ue.uex_Uib_Reason_Code = p->reason;
        if p->newsev < 0 then;
        else
          mf.uex_Mfa_New_Severity = p->newsev;
      end;
  end;


  exitterm:
    proc ( ue, ta  )
    options( nodescriptor );

    dcl 1 ue like uex_Uib byaddr;
    dcl 1 ta like uex_Tsa byaddr;

    ue.uex_Uib_return_Code = 0;
    ue.uex_Uib_reason_Code = 0;

  end;

end pack;

//*********************************************************************
//* link the user exit
//*********************************************************************
//LKEDEXIT EXEC PGM=IEWL,PARM='XREF,LIST,LET,DYNAM=DLL',
//    COND=(9,LT,PLIEXIT),REGION=5000K
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD  DD DSN=&&EXITLIB(IBMUEXIT),DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(7,1,1)),DSNTYPE=LIBRARY
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(3,1)),
//            DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=X
//SYSDEFSD DD DUMMY
//SYSLIN    DD DSN=&&LOADSET,DISP=SHR
//         DD DDNAME=SYSIN
//LKED.SYSIN     DD *
  ENTRY IBMUEXIT
```

Sample JCL to compile, link, and invoke the user exit (continued)

```
//*********************************************************************
//* compile main
//*********************************************************************
//PLI EXEC PGM=IBMZPLI,PARM='F(I),EXIT',
//            REGION=256K
//STEPLIB   DD  DSN=&&EXITLIB,DISP=SHR
//          DD DSN=IBMZ.V5R1M0.SIBMZCMP,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET2,DISP=(MOD,PASS),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*process;
 MainFet: Proc Options(Main);
   /* the exit will suppress the message for the next dcl */
   dcl one_byte_integer  fixed bin(7);
 End ;
//*
//SYSUEXIT DD  DISP=SHR,DSN=hlq.some.user.exit.input.file *
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-------------------------------
  'IBM'     1042       -1          1      String spans multiple lines
  'IBM'     1044       -1          1      FIXED BIN 7 mapped to 1 byte
```

Sample JCL to compile, link, and invoke the user exit (continued)

Fetching PL/I MAIN routines in 31-bit applications

In an Enterprise PL/I application, you can also fetch a PL/I MAIN program. When a FETCH of a PL/I MAIN program occurs, a child enclave is created.

You must follow these rules:

- You cannot pass any runtime options to a fetched MAIN program in the parameter string.
- If the SYSTEM(MVS) compiler option is specified, you can pass an arbitrary parameter list, but if the parameter is anything other than a single CHAR VARYING string, the compiler flags the MAIN routine with a warning message.
- You must not specify OPTIONS(ASM) or OPTIONS(NODESCRIPTOR) in the ENTRY declaration for the fetched MAIN routine in the fetching program.
- Avoid passing runtime options because attempts to parse them might produce LE informational messages regarding invalid runtime options. If NOEXECOPS is specified in the fetched MAIN routine, the passed char varying string is not parsed for the runtime options.
- If no parameters are passed to the fetched MAIN program, the fetching program should either specify OPTIONS(LINKAGE(SYSTEM)) in its ENTRY declaration for the fetched MAIN routine, or be compiled with DEFAULT(LINKAGE(SYSTEM)).

Examples

Here is the sample of the PL/I fetched MAIN program:

```
FMAIN: proc(parm) options(main,noexecops );
  DCL parm char(*) var;
  DCL SYSPRINT print;
  DCL PLIXOPT CHAR(11) VAR INIT('RPTOPTS(ON)')
```

```
      STATIC EXTERNAL;
   Put skip list("FMAIN parm: "|| parm);
   Put skip list("FMAIN finished ");
End FMAIN;
```

Here is the sample of the PL/I MAIN program that fetches another PL/I MAIN program:

```
MainFet: Proc Options(main);
    Dcl Parm char(1000) var;
    Dcl SYSPRINT print;
    Dcl Fmain entry(char(*) var) ;
    Put skip list("MainFet: start ");
    Parm = 'local-parm';
    Put skip list("MainFet parm: "|| Parm);
    Fetch Fmain;
    Call Fmain(Parm);
    Release Fmain;
    Put skip list("MainFet:testcase finished ");
End;
```

Fetching z/OS C routines in 31-bit applications

Unless the NORENT option has been specified, the ENTRY declaration in the routine that fetches a z/OS C routine must not specify OPTIONS(COBOL) or OPTIONS(ASM)—these should be specified only for COBOL or ASM routines not linked as DLLs.

The z/OS C documentation provides instructions on how to compile and link a z/OS C DLL.

Fetching assembler routines in 31-bit applications

Unless the NORENT option has been specified, the ENTRY declaration in the routine that fetches an assembler routine must specify OPTIONS(ASM).

When fetching data-only assembler modules, you must use the FETCH A SET(P) construct to ensure that pointer P is set correctly.

Invoking MAIN under TSO/E

If you compile your MAIN program with the SYSTEM(MVS) option, you can invoke the program by using the TSO CALL command or as a TSO command processor. Both the runtime options and parameters can be passed in the same way as under MVS batch.

For example, if the MAIN program TSOARG1 was link-edited as a member of a data set named `userid.TEST.load`, it can be invoked as follows:

```
   CALL TEST(TSOARG1) 'RPTSTG(ON),TRAP(ON)/THIS IS MY ARGUMENT'
or
   CALL TEST(TSOARG1) '/THIS IS MY ARGMENT'
or
   TSOARG1 TRAP(ON)/THIS IS MY ARGUMENT
or
   TSOARG1 /THIS IS MY ARGUMENT
```

Note: The data set containing TSOARG1 (`userid.TEST.load`) must be in the standard TSO program search list to run the program without using the CALL statement. This can be accomplished by issuing the TSO command:

```
TSOLIB ACTIVATE DSN('userid.TEST.load')
```

However, if you compile your MAIN program with the SYSTEM(TSO) option, the program will be passed a pointer to the Command Processor Parameter List (CPPL). In this case, NOEXECOPS is in effect. Your program can be invoked as a TSO command, but it cannot be invoked by a TSO CALL statement. See the following example:

```
TSOARG2 This is my argument
```

The program in Figure 19 uses the SYSTEM(TSO) interface to address and display the program arguments from the CPPL.

```
*process system(tso);
 tsoarg2: proc (cppl_ptr) options(main);
   dcl cppl_ptr pointer;
   dcl 1 cppl based(cppl_ptr),
     2 cpplcbuf   pointer,
     2 cpplupt    pointer,
     2 cpplpscb   pointer,
     2 cpplect    pointer;
   dcl 1 cpplbuf based(cpplcbuf),
     2 len    fixed bin(15),
     2 offset fixed bin(15),
     2 argstr char(1000);
   dcl my_argument char(1000) varying;
   dcl my_argument_len fixed bin(31);
   dcl length builtin;

   my_argument_len = len - offset - 4;
   if my_argument_len = 0 then
     my_argument = '';
   else
     my_argument =  substr(argstr,offset + 1, my_argument_len);
   display('Program args: ' || my_argument);
 end tsoarg2;
```

Figure 19. Sample program to display program arguments from the CPPL under TSO when using SYSTEM(STD) option

Whether your MAIN program is invoked as a command or by CALL, you can always specify runtime options through the PLIXOPT string.

Invoking MAIN under z/OS UNIX

Under z/OS UNIX, you can compile a MAIN program with the SYSTEM(MVS) or SYSTEM(OS) option; however, the number and format of the parameters passed to the program differs based on the option that you use.

If you compile a MAIN program with the SYSTEM(MVS) option, the program will be passed, as usual, one CHARACTER VARYING string containing the parameters specified when it was invoked.

However, if you compile a MAIN program with the SYSTEM(OS) option, the program will be passed 7 parameters as specified in the z/OS UNIX manuals. These 7 parameters include the following:

- The argument count (which includes the name of the executable as the first "argument")
- The address of an array of addresses of the arguments

- The address of an array of addresses of the arguments as null-terminated character strings
- The count of environment variables set
- The address of an array of addresses of the lengths of the environment variables
- The address of an array of addresses of the environment variables as null-terminated character strings

The program in Figure 20 uses the SYSTEM(OS) interface to address and display the individual arguments and environment variables.

```
*process display(std) system(os);

 sayargs:
  proc(argc, pArgLen, pArgStr, envc, pEnvLen, pEnvStr, pParmSelf)
  options( main, noexecops );

    dcl argc                fixed bin(31) nonasgn byaddr;
    dcl pArgLen             pointer nonasgn byvalue;
    dcl pArgStr             pointer nonasgn byvalue;
    dcl envc                fixed bin(31) nonasgn byaddr;
    dcl pEnvLen             pointer nonasgn byvalue;
    dcl pEnvStr             pointer nonasgn byvalue;
    dcl pParmSelf           pointer nonasgn byvalue;

    dcl q(4095)             pointer based;
    dcl bxb                 fixed bin(31) based;
    dcl bcz                 char(31) varz based;

    display( 'argc = ' || argc );
    do jx = 1 to argc;
      display( 'pargStr(jx) =' || pArgStr->q(jx)->bcz );
    end;
    display( 'envc = ' || envc );
    do jx = 1 to envc;
      display( 'pEnvStr(jx) =' || pEnvStr->q(jx)->bcz );
    end;

  end;
```

Figure 20. Sample program to display z/OS UNIX arguments and environment variables

Chapter 6. Link-editing and running for 64-bit programs

After compilation with LP(64), your 64-bit program consists of one or more object modules that contain unresolved references to each other, as well as references to the Language Environment runtime library. These references are resolved during link-editing (statically) or during execution (dynamically).

After you compile your PL/I program, the next step is to link and run your program with test data to verify that it produces the results that you expect.

Language Environment provides the runtime environment and services that you need to execute your program. For instructions on linking and running PL/I and all other Language Environment-conforming language programs, see the *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*. For information about migrating your existing PL/I programs to Language Environment, see the *Enterprise PL/I for z/OS Compiler and Run-Time Migration Guide*.

Link-edit considerations for 64-bit programs

You must use a PDSE for your linker output.

Using the binder in 64-bit programs

You must place the binder output into a PDSE.

When linking a DLL, specify any needed definition side-decks during the bind step. At a minimum, the PL/I side-deck IBMPQV11 and the C side-deck CELQS003 must be included. These side-decks can be found in the SCEELIB data set.

Using the ENTRY card in 64-bit programs

If you are building a module that will be fetched and that has an Enterprise PL/I routine as its entry point, the ENTRY card must specify CELQSTRT.

Runtime considerations for 64-bit programs

You can specify runtime options as parameters passed to the program initialization routine. You can also specify runtime options in the PLIXOPT variable. It might also prove beneficial, from a performance standpoint, if you alter your existing programs by using the PLIXOPT variable to specify your runtime options and recompiling your programs.

For a description of using PLIXOPT, see the *z/OS Language Environment Programming Guide*.

To simplify input/output at the terminal, various conventions have been adopted for stream files that are assigned to the terminal. Three areas are affected:

1. Formatting of PRINT files
2. The automatic prompting feature
3. Spacing and punctuation rules for input

© Copyright IBM Corp. 1999, 2017

Note: No prompting or other facilities are provided for record I/O at the terminal, so you are strongly advised to use stream I/O for any transmission to or from a terminal.

Formatting conventions for PRINT files for 64-bit programs

When a PRINT file is assigned to the terminal, it is assumed that it will be read as it is being printed. Spacing is therefore reduced to a minimum to reduce printing time.

The following rules apply to the PAGE, SKIP, and ENDPAGE keywords:

- PAGE options or format items result in three lines being skipped.
- SKIP options or format items larger than SKIP (2) result in three lines being skipped. SKIP (2) or less is treated in the usual manner.
- The ENDPAGE condition is never raised.

Automatic prompting

When the program requires input from a file that is associated with a terminal, it issues a prompt. This takes the form of printing a colon on the next line and then skipping to column 1 on the line following the colon. This gives you a full line to enter your input.

See the following example:

```
:
(space for entry of your data)
```

This type of prompt is referred to as a primary prompt.

Overriding automatic prompting

You can override the primary prompt by making a colon the last item in the request for the data. You cannot override the secondary prompt.

For example, the following two PL/I statements result in the terminal displaying output shown in Figure 16 on page 170.

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));
```

```
ENTER TIME OF PERIHELION
:         (automatic prompt)
(space for entry of data)
```

Figure 21. Output with automatic prompt

However, if the first statement has a colon at the end of the output as follows, the automatic prompt is overridden; Figure 17 on page 170 shows the sequence that is displayed on the terminal.

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

Figure 22. Output with no automatic prompt

Note: The override remains in force for only one prompt. You will be automatically prompted for the next item unless the automatic prompt is again overridden.

Punctuating long input lines

To transmit data that requires two or more lines of space at the terminal as one data item, you must use an SBCS hyphen as the line continuation character. Type an SBCS hyphen as the last character in each line except the last line.

For example, you must enter data as follows to transmit this sentence: `this data must be transmitted as one unit.`

```
:'this data must be transmitted -
+:as one unit.'
```

Transmission does not occur until you press ENTER after `unit.'`. The hyphen is removed. The item transmitted is called a *logical line*.

Note: To transmit a line whose last data character is a hyphen or a PL/I minus sign, enter two hyphens at the end of the line, followed by a null line as the next line. See the following example:

```
xyz--
(press ENTER only on this line)
```

Punctuating GET LIST and GET DATA statements

For GET LIST and GET DATA statements, a comma is added to the end of each logical line transmitted from the terminal, if the programmer omits it. Thus there is no need to enter blanks or commas to delimit items if they are entered on separate logical lines.

Given the PL/I statement `GET LIST(A,B,C);`, you can enter data as follows at the terminal:

```
:1
+:2
+:3
```

This rule also applies when you enter character-string data. Therefore, a character string must transmit as one logical line. Otherwise, commas are placed at the break points. For example, suppose you enter the following data:

```
:'COMMAS SHOULD NOT BREAK
+:UP A CLAUSE.'
```

The resulting string is `COMMAS SHOULD NOT BREAK, UP A CLAUSE.`

The comma is not added if a hyphen was used as a line continuation character.

Automatic padding for GET EDIT

For a GET EDIT statement, you do not need to enter blanks at the end of the line. The data will be padded to the specified length.

For example, given the PL/I statement `GET EDIT (NAME) (A(15));`, you can enter the 5 characters SMITH, and the data will be padded with ten blanks so that the program receives the fifteen characters:

```
'SMITH          '
```

Note: A single data item must transmit as a logical line. Otherwise, the first line transmitted will be padded with the necessary blanks and taken as the complete data item.

Use of SKIP for terminal input

All uses of SKIP for input are interpreted as SKIP(1) when the file is allocated to the terminal. SKIP(1) is treated as an instruction to ignore all unused data on the currently available logical line.

ENDFILE

You can enter end-of-file at the terminal by typing a logical line that consists of the two characters `/*`.

Any further attempts to use the file without closing it result in the ENDFILE condition being raised.

SYSPRINT considerations for 64-bit programs

For 64-bit programs, SYSPRINT is equated to the C stdout file. In addition, shared SYSPRINT is not supported.

The LINESIZE of SYSPRINT cannot be greater than 132 (the largest value allowed by C).

Using FETCH in your routines in 64-bit applications

In Enterprise PL/I, you can fetch routines compiled by PL/I, C, or the assembler.

Fetching Enterprise PL/I routines in 64-bit applications

A fetched module can perform FETCH, I/O, ALLOCATE, and FREE operations. However, a few restrictions still apply.

A fetched module can perform the following operations:

- Fetch other modules.
- Perform any I/O operations on any PL/I file. The file can be opened either by the fetched module, by the main module, or by some other fetched module.
- ALLOCATE and FREE its own CONTROLLED variables.

There are, however, a few restrictions on an Enterprise PL/I module that is to be fetched:

1. You must specify OPTIONS(FETCHABLE) on the PROCEDURE statement of the fetched routine if no ENTRY card is provided during the link-edit step.
2. The ENTRY card must specify CELQSTRT.
3. Because under LP(64) RENT is effectively always on, the ENTRY that is fetched must not be declared in the fetching module as OPTIONS(COBOL) or OPTIONS(ASM). If you want to avoid passing descriptors in this situation, you must specify the OPTIONS(NODESCRIPTOR) attribute on the ENTRY declaration.

4. An Enterprise PL/I routine cannot fetch itself.

Fetching PL/I MAIN routines in 64-bit applications

A 64-bit PL/I MAIN routine cannot fetch a 64-bit PL/I MAIN routine.

Fetching assembler routines in 64-bit applications

The ENTRY declaration in the routine that fetches an assembler routine must specify OPTIONS(ASM). The assembler routine must be linked with AMODE=64. LINKAGE(SYSTEM) should be specified.

When fetching data-only assembler modules, you must use the FETCH A SET(P) construct to ensure that pointer P is set correctly.

Invoking MAIN under TSO/E

If you compile your MAIN program with the SYSTEM(MVS) option, you can invoke the program by using the TSO CALL command or as a TSO command processor. Both the runtime options and parameters can be passed in the same way as under MVS batch.

For example, if the MAIN program TSOARG1 was link-edited as a member of a data set named `userid.TEST.load`, it can be invoked as follows:

```
   CALL TEST(TSOARG1) 'RPTSTG(ON),TRAP(ON)/THIS IS MY ARGUMENT'
 or
   CALL TEST(TSOARG1) '/THIS IS MY ARGMENT'
 or
   TSOARG1 TRAP(ON)/THIS IS MY ARGUMENT
 or
   TSOARG1 /THIS IS MY ARGUMENT
```

Note: The data set containing TSOARG1 (`userid.TEST.load`) must be in the standard TSO program search list to run the program without using the CALL statement. This can be accomplished by issuing the TSO command:

```
TSOLIB ACTIVATE DSN('userid.TEST.load')
```

However, if you compile your MAIN program with the SYSTEM(TSO) option, the program will be passed a pointer to the Command Processor Parameter List (CPPL). In this case, NOEXECOPS is in effect. Your program can be invoked as a TSO command, but it cannot be invoked by a TSO CALL statement. See the following example:

```
TSOARG2 This is my argument
```

The program in Figure 23 on page 192 uses the SYSTEM(TSO) interface to address and display the program arguments from the CPPL.

```
*process system(tso);
 tsoarg2: proc (cppl_ptr) options(main);
   dcl cppl_ptr pointer;
   dcl 1 cppl based(cppl_ptr),
     2 cpplcbuf   pointer,
     2 cpplupt    pointer,
     2 cpplpscb   pointer,
     2 cpplect    pointer;
   dcl 1 cpplbuf based(cpplcbuf),
     2 len    fixed bin(15),
     2 offset fixed bin(15),
     2 argstr char(1000);
   dcl my_argument char(1000) varying;
   dcl my_argument_len fixed bin(31);
   dcl length builtin;

   my_argument_len = len - offset - 4;
   if my_argument_len = 0 then
     my_argument = '';
   else
     my_argument =  substr(argstr,offset + 1, my_argument_len);
   display('Program args: ' || my_argument);
 end tsoarg2;
```

Figure 23. Sample program to display program arguments from the CPPL under TSO when using SYSTEM(STD) option

Whether your MAIN program is invoked as a command or by CALL, you can always specify runtime options through the PLIXOPT string.

Invoking MAIN under z/OS UNIX

Under z/OS UNIX, you can compile a MAIN program with the SYSTEM(MVS) or SYSTEM(OS) option; however, the number and format of the parameters passed to the program differs based on the option that you use.

If you compile a MAIN program with the SYSTEM(MVS) option, the program will be passed, as usual, one CHARACTER VARYING string containing the parameters specified when it was invoked.

However, if you compile a MAIN program with the SYSTEM(OS) option, the program will be passed 7 parameters as specified in the z/OS UNIX manuals. These 7 parameters include the following:

- The argument count (which includes the name of the executable as the first "argument")
- The address of an array of addresses of the arguments
- The address of an array of addresses of the arguments as null-terminated character strings
- The count of environment variables set
- The address of an array of addresses of the lengths of the environment variables
- The address of an array of addresses of the environment variables as null-terminated character strings

The program in Figure 24 on page 193 uses the SYSTEM(OS) interface to address and display the individual arguments and environment variables.

```
*process display(std) system(os);

 sayargs:
  proc(argc, pArgLen, pArgStr, envc, pEnvLen, pEnvStr, pParmSelf)
  options( main, noexecops );

   dcl argc                fixed bin(31) nonasgn byaddr;
   dcl pArgLen             pointer nonasgn byvalue;
   dcl pArgStr             pointer nonasgn byvalue;
   dcl envc                fixed bin(31) nonasgn byaddr;
   dcl pEnvLen             pointer nonasgn byvalue;
   dcl pEnvStr             pointer nonasgn byvalue;
   dcl pParmSelf           pointer nonasgn byvalue;

   dcl q(4095)             pointer based;
   dcl bxb                 fixed bin(31) based;
   dcl bcz                 char(31) varz based;

   display( 'argc = ' || argc );
   do jx = 1 to argc;
     display( 'pargStr(jx) =' || pArgStr->q(jx)->bcz );
   end;
   display( 'envc = ' || envc );
   do jx = 1 to envc;
     display( 'pEnvStr(jx) =' || pEnvStr->q(jx)->bcz );
   end;

 end;
```

Figure 24. Sample program to display z/OS UNIX arguments and environment variables

Chapter 7. Considerations for developing 64-bit applications

You can use Enterprise PL/I to develop 31-bit or 64-bit applications. For your applications to support the 64-bit environment, you might need to adapt your code as appropriate. This section describes considerations in development and compilation that you must take into account.

Using compiler options to build 64-bit applications

To compile code for 64-bit applications, you must use the LP(64) compiler option. You must be aware that under LP(64), some compiler options or suboptions are not supported, and that some options or suboptions are ignored during compilation.

Note the following compiler options when you compile code under LP(64):

"BACKREG" on page 10
This option is ignored.

"CEESTART" on page 13
This option is ignored.

"CHECK" on page 14
The STORAGE suboption is not supported under LP(64). This means that you cannot use the following built-in functions:

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

"CMPAT" on page 15
This option is ignored; effectively, CMPAT(V3) is always on under LP(64).

"COMMON" on page 16
This option is ignored.

"DEFAULT" on page 22
The following suboptions are ignored under LP(64):

- LINKAGE
- NULL370 | NULLSYS

"RENT" on page 68
This option is ignored; effectively, RENT is always on under LP(64).

"STDSYS" on page 84
This option is ignored; effectively, STDSYS is always on under LP(64).

"WRITABLE" on page 93
This option is ignored.

Related information:

"LP" on page 49
The LP option specifies whether the compiler generates 31-bit code or 64-bit code. It also determines the default size of POINTER and HANDLE and related variables.

© Copyright IBM Corp. 1999, 2017

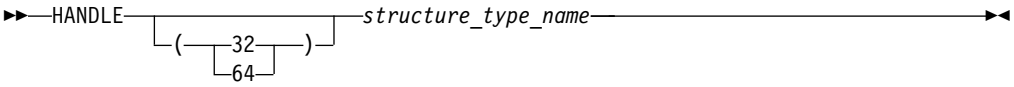
Using attributes HANDLE and POINTER under LP(64)

The default size and alignment of HANDLE and POINTER under LP(64) are different from the default under LP(32).

HANDLE attribute

Under LP(32), the default is HANDLE(32); under LP(64), the default is HANDLE(64).

Syntax



A HANDLE(32) is four bytes in size and by default fullword-aligned.

A HANDLE(64) is eight bytes in size and by default doubleword-aligned.

HANDLE(64) is valid only under LP(64).

Assigning a HANDLE(32) to a HANDLE(64) is always valid; the reverse is valid only if the first four bytes of the HANDLE(64) are zero.

Because of the change in the size and alignment of handles, structures that contain them might have padding bytes.

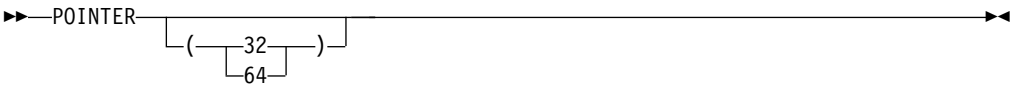
Related information:

"LP" on page 49
The LP option specifies whether the compiler generates 31-bit code or 64-bit code. It also determines the default size of POINTER and HANDLE and related variables.

POINTER attribute

Under LP(32), the default is POINTER(32); under LP(64), the default is POINTER(64).

Syntax



A POINTER(32) is four bytes in size and by default fullword-aligned.

A POINTER(64) is eight bytes in size and by default doubleword-aligned.

POINTER(64) is valid only under LP(64).

Assigning a POINTER(32) to a POINTER(64) is always valid; the reverse is valid only if the first four bytes of the POINTER(64) are zero.

Because of the change in the size and alignment of pointers, structures that contain them might have padding bytes.

Related information:

"LP" on page 49
The LP option specifies whether the compiler generates 31-bit code or 64-bit code. It also determines the default size of POINTER and HANDLE and related variables.

Using ENTRY variables under LP(64)

Under LP(64), all ENTRY variables, whether they have the LIMITED attribute or not, are eight bytes in size and are by default doubleword-aligned. Therefore, a structure that contains an ENTRY variable might now have padding bytes.

Using built-in functions under LP(64)

When you develop 64-bit applications, you must be aware that the argument and return types of some built-in functions are different.

Built-in functions that return a FIXED BIN(63) value under LP(64)

The following built-in functions return a FIXED BIN(63) value under LP(64), but under LP(32) they return a FIXED BIN(31) value.

AUTOMATIC
AVAILABLEAREA
BASE64DECODE16
BASE64DECODE8
BASE64ENCODE16
BASE64ENCODE8
CURRENTSTORAGE
FILEID
FILEREAD
FILESEEK
FILETELL
HEXDECODE
HEXDECODE8
JSONGETARRAYEND
JSONGETARRAYSTART
JSONGETCOLON
JSONGETCOMMA
JSONGETMEMBER
JSONGETOBJECTEND
JSONGETOBJECTSTART
JSONGETVALUE
JSONPUTARRAYEND
JSONPUTARRAYSTART
JSONPUTCOLON
JSONPUTCOMMA
JSONPUTMEMBER
JSONPUTOBJECTEND
JSONPUTOBJECTSTART
JSONPUTVALUE
JSONVALID
MEMCONVERT
MEMCU12
MEMCU14
MEMCU21
MEMCU24
MEMCU12
MEMCU41
MEMCU42
MEMINDEX
MEMSEARCH
MEMSEARCHR
MEMVERIFY
MEMVERIFYR
LOCATION
LOCSTG
OFFSETDIFF
POINTERDIFF
STORAGE
WHITESPACECOLLAPSE
WHITESPACEREPLACE
XMLCHAR
XMLCLEAN

Built-in functions whose integer arguments are converted to FIXED BIN(63) under LP(64)

The following built-in functions have one or more arguments that represent the size of a piece of storage. These arguments are converted to FIXED BIN(63) under LP(64), if necessary.

ALLOCATE
BASE64DECODE8
BASE64DECODE16
BASE64ENCODE8
BASE64ENCODE16
CHECKSUM
COMPARE
FILEWRITE
HEXDECODE
HEXDECODE8
JSONGETARRAYEND
JSONGETARRAYSTART
JSONGETCOLON
JSONGETCOMMA
JSONGETMEMBER
JSONGETOBJECTEND
JSONGETOBJECTSTART
JSONGETVALUE
JSONPUTARRAYEND
JSONPUTARRAYSTART
JSONPUTCOLON
JSONPUTCOMMA
JSONPUTMEMBER
JSONPUTOBJECTEND
JSONPUTOBJECTSTART
JSONPUTVALUE
JSONVALID
MEMCONVERT
MEMCU12
MEMCU14
MEMCU21
MEMCU24
MEMCU41
MEMCU42
MEMINDEX
MEMSEARCH
MEMSEARCHR
MEMVERIFY
MEMVERIFYR
PLIASCII
PLIEBCDIC
PLIFILL
PLIMOVE
PLIOVER
PLISAXA
PLISAXB
PLISAXC
PLISAXD
PLITRAN11
PLITRAN12
PLITRAN21
PLITRAN22
WHITESPACECOLLAPSE
WHITESPACEREPLACE
XMLCHAR
XMLCLEAN

Built-in functions that are not supported under LP(64)

Under LP(64), you cannot use the following built-in functions because the STORAGE suboption of the CHECK compiler option is not supported:

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

Considerations for SQL programs

To use the same source to develop 32-bit and 64-bit applications, it is recommended that you declare `sqlda` or `sqlda2` by using the `EXEC SQL INCLUDE` statement, and that the `sqldabc` field be set to use `stg(sqlvar(1))` as the size of `sqlvar`. If your SQL program is not following these recommendations, you might need to change your code to develop a 64-bit application.

If needed, you can make these changes to existing 32-bit SQL programs without changing their behavior. These changes can make these programs easier to port.

Declaring sqlda

If `sqlda` is not declared with the `EXEC SQL INCLUDE sqlda` statement, you must change the declaration for `sqlda` by adding the following field after the `sqllen` field:

```
char( length(hex(sysnull()))/2 - 4 ),
```

For example, assume that in your program the sqlda structure is declared as follows:

```
dcl
      1 fsqlda based(fsqldaptr),
        2 sqldaid      char(8),
        2 sqldabc      fixed bin(31),
        2 sqln         fixed bin(15),
        2 sqld         fixed bin(15),
        2 sqlvar(fsqlsize refer(sqln)),
          3 sqltype    fixed bin(15),
          3 sqllen     fixed bin(15),
          3 sqldata    pointer,
          3 sqlind     pointer,
          3 sqlname    char(30) var;
```

You must change the declaration as follows:

```
dcl
      1 fsqlda based(fsqldaptr),
        2 sqldaid      char(8),
        2 sqldabc      fixed bin(31),
        2 sqln         fixed bin(15),
        2 sqld         fixed bin(15),
        2 sqlvar(fsqlsize refer(sqln)),
          3 sqltype    fixed bin(15),
          3 sqllen     fixed bin(15),
          3 *          char( length(hex(sysnull()))/2 - 4 ),
          3 sqldata    pointer,
          3 sqlind     pointer,
          3 sqlname    char(30) var;
```

Declaring sqlda2

If sqlda2 is not declared with the EXEC SQL INCLUDE sqlda2 statement, you must change the declaration for sqlda2 by adding the following field after the sqlrsvdl field:

```
       char( length(hex(sysnull())) - 8 ),
```

For example, assume that in your program the sqlda2 structure is declared as follows:

```
dcl
      1 fsqlda2 based(fsqldaptr),
        2 sqldaid2     char(8),
        2 sqldabc2     fixed bin(31),
        2 sqln2        fixed bin(15),
        2 sqld2        fixed bin(15),
        2 sqlvar2(fsqlsize refer(sqln2)),
          3 sqlbiglen,
            4 sqllongl fixed bin(31),
            4 sqlrsvdl fixed bin(31),
          3 sqldatal   pointer,
          3 sqltname   char(30) var;
```

You must change the declaration as follows:

```
dcl
      1 fsqlda2 based(fsqldaptr),
        2 sqldaid2     char(8),
        2 sqldabc2     fixed bin(31),
        2 sqln2        fixed bin(15),
        2 sqld2        fixed bin(15),
        2 sqlvar2(fsqlsize refer(sqln2)),
          3 sqlbiglen,
            4 sqllongl fixed bin(31),
```

```
  4 sqlrsvdl fixed bin(31),
3 *          char( length(hex(sysnull())) - 8 ),
3 sqldatal   pointer,
3 sqltname   char(30) var;
```

Setting sqldabc

If the `sqldabc` field is set to 44 as the size of `sqlvar`, you must set it to use `stg(sqlvar(1))` instead.

For example, the statement `sqlda.sqldabc = 16 + ( 44 * sqlda.sqld );` should be changed to `sqlda.sqldabc = 16 + ( stg(sqlda.sqlvar(1)) * sqlda.sqld );`.

You can code as follows:

```
SQLDABC = LEN_SQLDA + SQLN * LEN_SQLVAR;
```

You must change the declaration for LEN_SQLVAR as follows:

Original declaration:
```
DCL LEN_SQLVAR FIXED BIN(15) VALUE( 44 );
```

Updated declaration:
```
DCL LEN_SQLVAR FIXED BIN(15) VALUE( STG(SQLDA.SQLVAR(1)) );
```

Communicating with 31-bit routines

Language Environment does not support mixing 64-bit and 31-bit programs in the same application. For example, you cannot call a PL/I program compiled with LP(32) from a PL/I program compiled with LP(64).

To facilitate communicating with *non-Language Environment based* 31-bit routines, a generic interface is provided to load, call and release a 31-bit routine. A function to retrieve the entry point address for the loaded module is also provided.

Invoke IBMPC32I to load a 31-bit routine. It takes a pointer to an 8-bit character string that holds the name of the module as input and returns a file handle to be used for the call and release functions.

```
dcl <load32>   ext( "_IBMPC32I" )
              entry( char(8) byaddr inonly )
              returns( pointer byvalue )
              options( nodescriptor linkage(optlink) );
```

Invoke IBMPC32C to call a 31-bit routine. It takes two parameters as input: the first parameter is the file handle returned from the IBMPC32I call, and the second parameter is the user parameter list to the 31-bit routine. Note that the user parameter list has to reside in below-the-bar storage. IBMPC32C returns the register 15 value from the called 31-bit routine.

```
dcl <call32>  ext( "_IBMPC32C" )
             entry( pointer byvalue, pointer byvalue )
             returns( fixed bin(31) byvalue )
             options( nodescriptor linkage(optlink) );
```

Invoke IBMPC32T to release a 31-bit routine. It takes the file handle as a parameter.

```
dcl <rel32>    ext( "_IBMPC32T" )
              entry( pointer byvalue )
              options( nodescriptor linkage(optlink) );
```

Invoke IBMPC32E to get the entry point address of the loaded module. It takes the file handle as a parameter and returns the entry point address of the loaded module. Note that it is not valid to branch to this address directly.

```
dcl <epa32>   ext( "_IBMPC32E" )
              entry( pointer byvalue )
              returns( pointer byvalue )
              options( nodescriptor linkage(optlink) );
```

The following code fragment example shows how the above interfaces can be used. DSNALI is a 31-bit DB2 facility.

```
....

dcl load32      ext( "_IBMPC32I" )
                     entry( char(8) byaddr inonly )
                     returns( pointer byvalue )
                     options( nodescriptor );

dcl call32     ext( "_IBMPC32C" )
                    entry( pointer byvalue, pointer byvalue )
                    returns( fixed bin(31) byvalue )
                    options( nodescriptor );

dcl  file_pointer   pointer;
dcl  plist_pointer pointer;
dcl 1 dsnali_plist based(plist_pointer),
3 args_list(10)        pointer(32),
3 content union,
5 connect,
7 functioncode  char(12),
7 subsystemid  char(4),
7 tecb              fixed bin(31),
7 secb             fixed bin(31),
7 ribpointer       ptr(32),
7 returncode     fixed bin(31),
7 reasoncode    fixed bin(31),
5 open,
7 functioncode  char(12),
7 subsystemid   char(4),
7 planname       char(8),
7 returncode     fixed bin(31),
7 reasoncode   fixed bin(31);

  dcl lastargflag bit(1) based;

  ....

 plist_pointer = alloc31( stg(dsnali_plist) );
 file_pointer = load32( "DSNALI" );

 /* set up argument list for CONNECT function for DSNALI call */
 args_list( 1 ) = addr( connect.functioncode );
 args_list( 2 ) = addr( connect.subsystemid );
 args_list( 3 ) = addr( connect.tecb );
 args_list( 4 ) = addr( connect.secb );
 args_list( 5 ) = addr( connect.ribpointer );
 args_list( 6 ) = addr( connect.returncode ) ;
 args_list( 7 ) = addr( connect.reasoncode ) ;

 /* mark last argument */
 addr( args_list( 7 ) ).lastargflag = '1'b;

 /* set up values for CONNECT function parameters */
 connect.functioncode = 'CONNECT';
 connect.subsystemid = 'DB2S';
 connect.tecb = 0;
```

```
connect.secb = 0;
connect.ribpointer = sysnull;
connect.returncode = 0;
connect.reasoncode = 0;

/* invoke DSNALI with CONNECT function */
rc = call32( file_pointer, plist_pointer );

...

/* set up argument list for OPEN function for DSNALI call */
args_list( 1 ) = addr( open.functioncode );
args_list( 2 ) = addr( open.subsystemid );
args_list( 3 ) = addr( open.planname );
args_list( 4 ) = addr( open.returncode );
args_list( 5 ) = addr( open.reasoncode );

/* mark last argument */
addr( args_list( 5 ) ).lastargflag = '1'b;

/* set up values for OPEN function parameters */
open.functioncode = 'OPEN';
open.subsystemid = 'DB2S';
open.planname = 'TESTCASE';
open.returncode = 0;
open.reasoncode = 0;

/* invoke DSNALI with OPEN function */
rc = call32( file_pointer, plist_pointer ) ;

 ....
```

Part 2. Using I/O facilities

© Copyright IBM Corp. 1999, 2017

Chapter 8. Using data sets and files

This chapter describes how to allocate files and associate data sets with the files known within your program. It introduces the five major types of data sets, describes how they are organized and accessed, and helps you learn how to specify some of the file and data set characteristics.

Your PL/I programs process and transmit units of information called *records*. A collection of records is called a *data set*. Data sets are physical collections of information external to PL/I programs; they can be created, accessed, or modified by programs written in PL/I or other languages or by the utility programs of the operating system.

Your PL/I program recognizes and processes information in a data set by using a symbolic or logical representation of the data set called a *file*.

Note: INDEXED implies VSAM and is supported only under batch.

Note: Regional data sets are not supported for 64-bit programs in PL/I V5.1.

Allocating files

For any type of file, including sequential, VSAM, REGIONAL(1), and HFS, you can define the external name by using the following methods:

- In the MVS or TSO environment:
  - A *ddname* in the JCL
  - An environment variable name
  - The TITLE option of the OPEN statement
- In the z/OS UNIX System Services environment:
  - An environment variable name
  - The TITLE option of the OPEN statement

For HFS files under the batch environment, the following rules apply:

- If FILEDATA is not specified, the default setting is TEXT.
- If the record format or the ENVIRONMENT option is not specified, the default is V with the LF type.
- If FILEDATA is BINARY, only fixed-length files are valid. Error message MSGIBM0210S is issued when FILEDATA=BINARY is specified with varying-length files.
- For varying-length files, files are assumed to be TYPE=LF; that is, the records are delimited with the LF (x'15') character.
- For data files that contain delimiters, specify FILEDATA=TEXT, so the PL/I library can handle the delimiters properly.

Dynamic allocation

A PL/I program dynamically allocates the file by using the attributes that are specified by the environment variable or the TITLE option.

© Copyright IBM Corp. 1999, 2017

To use PL/I dynamic allocation, you must specify the file names by using the DSN() format for MVS data sets or the PATH() format for HFS files. All options and attributes must be in uppercase, except for the **pathname** suboption of the PATH option, which is case-sensitive. Do not use temporary data set names in the DSN() option. See the following examples to specify file names:

```
OPEN FILE(FILEIN) TITLE('DSN(USER.FILE.EXT),SHR');
OPEN FILE(FILEIN) TITLE('PATH(/usr/FILE.EXT)');

EXPORT DD_FILE="DSN(USER.FILE.EXT),SHR"
EXPORT DD_FILE="PATH(/usr/FILE.EXT)"
```

The following rules of precedence apply in determining when the dynamic allocation takes place:

1. If one of the following DD statements exists for the file, it is used. This rule is not valid in the z/OS UNIX System Services environment.
   - JCL DD
   - TSO ALLOCATE
   - User-initiated dynamic allocation
2. If a DD statement does not exist for the file and the TITLE option is specified in the OPEN statement, the TITLE option is used by associating it with the external name for the file.
3. If a DD statement does not exist for the file and the TITLE option is not specified, but an environment variable exists for the file, the environment variable is used by associating it with the external name for the file.

For MVS data sets, the Enterprise PL/I run time checks the contents of the environment variable or TITLE option at each OPEN statement:

- If a file with the same external name was dynamically allocated by a previous OPEN statement, and the contents of the environment variable or the TITLE option have changed since that OPEN statement, the run time dynamically deallocates the previous allocation and reallocates the file by using the options that are currently set in the environment variable or specified in the TITLE option.
- If the contents of the environment variable or the TITLE option have not changed, the run time uses the current allocation without deallocating or reallocating.

For HFS files, the DD statement is deallocated and reallocated at each subsequent OPEN statement.

Note:

1. When you use the DSN() or the PATH() format specification for PL/I dynamic allocation, if a DD statement and the TITLE option of the OPEN statement are both specified for a file, the DD statement is used and the TITLE option is ignored.
2. Under the z/OS UNIX System Services environment, user-initiated dynamic allocation is not supported. If it is attempted, the UNDEFINEDFILE condition is raised, even when the TITLE option or the environment variable is used, because the external name is in use.

Associating data sets with files under z/OS

A file used within a PL/I program has a PL/I file name. The physical data set external to the program has a name by which it is known to the operating system: a *data set name* or *dsname*. In some cases the data set has no name; it is known to the system by the device on which it exists.

The operating system needs a way to recognize which physical data set is referred to by your program, so you must write a *data definition* or *DD* statement, external to your program, that associates the PL/I file name with a dsname.

For example, if you have the following file declaration in your program, you must create a DD statement with a *data definition name* (*ddname*) that matches the name of the PL/I file.

```
DCL STOCK FILE STREAM INPUT;
```

The DD statement specifies a physical data set name (dsname) and gives its characteristics:

```
//GO.STOCK  DD DSN=PARTS.INSTOCK, . . .
```

For more detail about writing DD statements, refer to the job control language (JCL) manuals for your system.

There is more than one way to associate a data set with a PL/I file. You associate a data set with a PL/I file by ensuring that the ddname of the DD statement that defines the data set is the same as one of the following:

- The declared PL/I file name
- The character-string value of the expression specified in the TITLE option of the associated OPEN statement.

You must choose your PL/I file names so that the corresponding ddnames conform to the following restrictions:

- If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that explicitly opens the file, the ddname uses the file name by default. If the file name is longer than 8 characters, the default ddname is composed of the first 8 characters of the file name.
- The character set of the JCL does not contain the break character (_). Consequently, this character cannot appear in ddnames. Do not use break characters among the first 8 characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters $, @, and #, however, are valid for ddnames, but the first character must be one of the letters A through Z.

Because external names are limited to 7 characters, an external file name of more than 7 characters is shortened into a concatenation of the first 4 and the last 3 characters of the file name. Such a shortened name is **not**, however, the name used as the ddname in the associated DD statement.

Consider the following statements:

```
1. OPEN FILE(MASTER);
2. OPEN FILE(OLDMASTER);
3. READ FILE(DETAIL) ...;
```

When statement number 1 is run, the file name MASTER is taken to be the same as the ddname of a DD statement in the current job step. When statement number 2 is run, the name OLDMASTE is taken to be the same as the ddname of a DD statement in the current job step. (The first 8 characters of a file name form the ddname. If OLDMASTER is an external name, it will be shortened by the compiler to OLDMTER for use within the program.) If statement number 3 causes implicit opening of the file DETAIL, the name DETAIL is taken to be the same as the ddname of a DD statement in the current job step.

In each of the above cases, a corresponding DD statement must appear in the job stream; otherwise, the UNDEFINEDFILE condition is raised. The three DD statements could start as follows:

1. `//MASTER   DD ...`
2. `//OLDMASTE DD ...`
3. `//DETAIL   DD ...`

If the file reference in the statement that explicitly or implicitly opens the file is not a file constant, the DD statement name **must** be the same as the value of the file reference. The following example illustrates how a DD statement should be associated with the value of a file variable:

```
DCL PRICES FILE VARIABLE,
    RPRICE FILE;
      PRICES = RPRICE;
      OPEN FILE(PRICES);
```

The DD statement should associate the data set with the file constant RPRICE, which is the value of the file variable PRICES:

```
//RPRICE DD DSNAME=...
```

Use of a file variable also allows you to manipulate a number of files at various times by a single statement. See the following example:

```
DECLARE F FILE VARIABLE,
        A FILE,
        B FILE,
        C FILE;
          .
          .
          .
     DO F=A,B,C;
        READ FILE (F) ...;
          .
          .
          .
     END;
```

The READ statement reads the three files A, B, and C, each of which can be associated with a different data set. The files A, B, and C remain open after the READ statement is executed in each instance.

The following OPEN statement illustrates use of the TITLE option:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

For this statement to be executed successfully, you must have a DD statement in the current job step with DETAIL1 as its ddname. It could start as follows:

```
//DETAIL1 DD DSNAME=DETAILA,...
```

Thus, you associate the data set DETAILA with the file DETAIL through the ddname DETAIL1.

Associating several files with one data set

You can use the TITLE option to associate two or more PL/I files with the same external data set, provided that the first file association is closed before a second file association is opened against the same TITLE name.

In the following example, INVNTRY is the name of a DD statement defining a data set to be associated with two files:

```
OPEN FILE (FILE1) TITLE('INVNTRY');
.....
CLOSE FILE (FILE1);
.....
OPEN FILE (FILE2) TITLE('INVNTRY');
```

If the file is not closed first before the second open is done, the UNDEFINEDFILE condition will be raised with a subcode1 value of 59, stating that an open was attempted against a file that was already open.

Associating several files with one data set

You can use the TITLE option to associate two or more PL/I files with the same external data set at the same time.

See the following example, where INVNTRY is the name of a DD statement defining a data set to be associated with two files:

```
OPEN FILE (FILE1) TITLE('INVNTRY');
OPEN FILE (FILE2) TITLE('INVNTRY');
```

If you do this, be careful. These two files access a common data set through separate control blocks and data buffers. When records are written to the data set from one file, the control information for the second file will not record that fact. Records written from the second file could then destroy records written from the first file. PL/I does not protect against data set damage that might occur. If the data set is extended, the extension is reflected only in the control blocks associated with the file that wrote the data; this can cause an abend when other files access the data set.

Associating several data sets with one file

You can use the TITLE option to associate several data sets with one file.

The file name can, at different times, represent entirely different data sets. Consider the following OPEN statement:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

For this statement to be executed successfully, a DD statement must be specified in the current job step with DETAIL1 as its ddname:

```
//DETAIL1 DD DSNAME=DETAILA,...
```

The file DETAIL1 is associated with the data set named in the DSNAME parameter of the DD statement DETAIL1. If you close and reopen the file, you can specify a different ddname in the TITLE option to associate the file with a different data set.

Using the TITLE option, you can choose dynamically, at open time, one among several data sets to be associated with a particular file name. Consider the following example:

```
DO IDENT='A','B','C';
   OPEN FILE(MASTER)
        TITLE('MASTER1'||IDENT);
    .
    .
    .
   CLOSE FILE(MASTER);
END;
```

In this example, when MASTER is opened during the first iteration of the do-group, the associated ddname is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the ddname. During the second iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the ddname MASTER1B. Similarly, during the final iteration of the do-group, MASTER is associated with the ddname MASTER1C.

Concatenating several data sets

For input only, you can concatenate two or more sequential or regional data sets (that is, link them so that they are processed as one continuous data set) by omitting the ddname from all but the first of the DD statements that describe the data sets.

For example, the following DD statements cause the data sets LIST1, LIST2, and LIST3 to be treated as a single data set for the duration of the job step in which the statements appear:

```
//GO.LIST DD DSNAME=LIST1,DISP=OLD
//         DD DSNAME=LIST2,DISP=OLD
//         DD DSNAME=LIST3,DISP=OLD
```

When read from a PL/I program, the concatenated data sets need not be on the same volume.

You cannot process concatenated data sets backward.

Accessing HFS files under z/OS

You can access HFS files from a batch program by specifying the HFS file name in the DD statement or in the TITLE option of the OPEN statement.

For example, to access the HFS file `/u/USER/sample.txt` by using the DD HFS, you can code the DD statement as follows:

```
   //HFS DD PATH='/u/USER/sample.txt',PATHOPTS=ORDONLY,DSNTYPE=HFS
```

To access the same file by using the TITLE option of the OPEN statement, you can code as follows:

```
   OPEN FILE(HFS) TITLE('//u/USER/sample.txt');
```

Note the two forward slashes in the TITLE option: the first indicates that what follows is a file name (rather than a DD name), and the second is the start of the fully qualified HFS file name. You must use fully qualified names when HFS files are referenced under batch, because there is no current directory that can be used to complete a file specification.

You can access HFS files from a batch program using PL/I dynamic allocation by specifying the HFS file name using one of the following methods:

- The DD statement
- The TITLE option of the OPEN statement
- The PUTENV built-in function
- The PLIXOPT string using the ENVAR option

The following example shows how to access HFS files by using these methods:

```
//HFS DD PATH='/u/USER/sample.txt',PATHOPTS=ORDONLY...

OPEN FILE(HFS) TITLE('PATH(/u/USER/sample.txt)');

xx = putenv('DD_HFS=/u/USER/sample.txt');

Dcl plixopt char(100) var ext static
    init('ENVAR("DD_HFS=PATH(/u/USER/sample.txt)")');
```

Note: To use PL/I dynamic allocation, specify the file names by using the DSN() format for MVS data sets, or the PATH() format for HFS files.

PL/I decides how to treat HFS files under batch in the following order:

1. If ENV(F) is specified in the file declaration, the file is assumed to consist of fixed length records.
2. If ENV(V) is specified in the file declaration, the file is assumed to consist of lf-delimited records.
3. If FILEDATA=BINARY is specified on the file's DD statement, the file is assumed to consist of fixed length records.
4. Otherwise the file is assumed to consist of lf-delimited records.

To access a fixed length z/OS file from UNIX, the record size of the file must be specified either in the ENVIRONMENT attribute of the file or in the TITLE option on the OPEN statement.

- If the file declaration does not contain ENV(F RECSIZE(...)), you must specify the data set name and these attributes in the TITLE option. For example, for a file of fixed length 80-byte records, you could specify a TITLE option as follows:

  ```
  '/dataset.name,type(fixed),recsize(80)'
  ```

- If the ENVIRONMENT attribute specifies only one of F or RECSIZE, you must specify the data set name and the omitted attribute in the TITLE option.
- If the ENVIRONMENT attribute specifies both F and RECSIZE, you need to specify only the data set name in the TITLE option.

Associating data sets with files under z/OS UNIX

A file used within a PL/I program has a PL/I file name. A data set also has a name by which it is known to the operating system.

PL/I needs a way to recognize the data sets to which the PL/I files in your program refer, so you must provide an identification of the data set to be used, or allow PL/I to use a default identification.

You can identify the data set explicitly using either an environment variable or the TITLE option of the OPEN statement.

To use PL/I dynamic allocation, you must specify the file names by using the DSN() format for MVS data sets, or the PATH() format for HFS files.

To access HFS files from z/OS UNIX program using PL/I dynamic allocation, you can specify the HFS filename by using one of the following methods:

- The TITLE option of the OPEN statement
- The PUTENV built-in function
- The PLIXOPT string using the ENVAR option
- The EXPORT statement

Using environment variables

Use the **export** command to establish an environment variable that identifies the data set to be associated with a PL/I file, and, optionally, to specify the characteristics of that data set. The information provided by the environment variable is called data definition (or DD) information.

These environment variable names have the form `DD_DDNAME`, where the *DDNAME* is the name of a PL/I file constant (or an *alternate DDNAME*, as defined below). If the filename refers to an HFS file, the filename must be properly qualified. Otherwise, the PL/I library assumes that the filename refers to an MVS data set.

Examples

- `declare MyFile stream output;`
  `export DD_MYFILE=/datapath/mydata.dat`

  where `/datapath/mydata.dat` refers to an HFS file. The filename is fully-qualified.
- `export DD_MYFILE=./mydata.dat`

  where `./mydata.dat` refers to an HFS file in the current directory.
- `export DD_MYFILE=mydata.dat`

  where `mydata.dat` refers to an MVS data set.

The following examples show that the HFS files are accessed when you use PL/I dynamic allocation:

```
export DD_HFS="PATH(/u/USER/sample.txt)"
export DD_FILE="DSN(USER.FILE.EXT),SHR"
```

If you are familiar with the IBM mainframe environment, you can think of the environment variable much like the following statement:

DD statement in z/OS
ALLOCATE statement in TSO

For more information about the syntax and options you can use with the DD_DDNAME environment variable, see “Specifying characteristics using DD_DDNAME environment variables” on page 214.

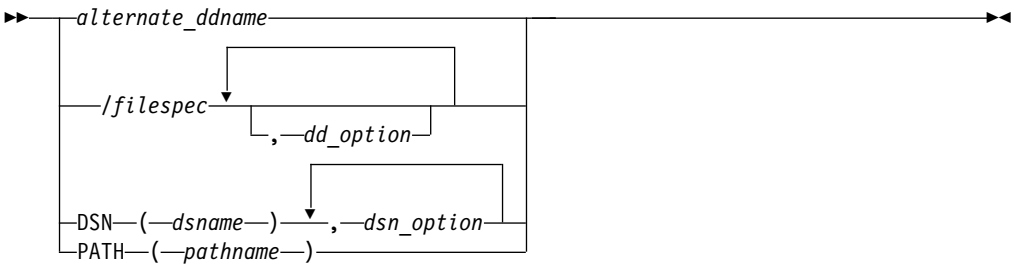
Under z/OS UNIX, where more types of varying length HFS files are supported than under batch, PL/I treats an HFS file as follows:

1. If ENV(F) is specified in the file declaration, the file is assumed to consist of fixed length records.
2. If a TYPE is specified in the file's EXPORT statement, the file is assumed to consist of records of that type.
3. Otherwise the file is assumed to consist of lf-delimited records.

Using the TITLE option of the OPEN statement

You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file, and, optionally, to provide additional characteristics of that data set.

The *expression* must yield a character string with the following syntax:



alternate_ddname
The name of an alternate DD_DDNAME environment variable

An alternate DD_DDNAME environment variable is not named after a file constant. For example, if you have a file named INVENTRY in your program and you establish two DD_DDNAME environment variables with the first named INVENTRY and the second named PARTS, you can associate the file with the second one by using this statement:

```
open file(Inventry) title('PARTS');
```

filespec
Any valid file specification on the system you are using

The maximum length of *filespec* is 1023 characters.

dd_option
One or more options allowed in a DD_DDNAME environment variable

For more information about the options of the DD_DDNAME environment variable, see "Specifying characteristics using DD_DDNAME environment variables" on page 214.

dsname
The fully qualified MVS data set name

dsn_option
One or more DSN options

For more information about the DSN options, see "Defining QSAM files using PL/I dynamic allocation" on page 262, "Defining REGIONAL(1) data sets using PL/I dynamic allocation" on page 281, and "Defining VSAM file using PL/I dynamic allocation" on page 291.

pathname
The fully qualified HFS path name

Here is an example of using the OPEN statement in this manner with a z/OS DSN:

```
open file(Payroll) title('/June.Dat,append(n),recsize(52)');
```

Note the required leading forward slash in the TITLE option. This leading forward slash indicates that what follows is a file name (rather than a DD name). In this case, `June.Dat` refers to an MVS data set.

If `June.Dat` is an HFS file, the OPEN statement example is as follows:

```
open file(Payroll) title('//u/USER/June.Dat,append(n),recsize(52)');
```

Note the two forward slashes in the TITLE option: the first indicates that what follows is a file name (rather than a DD name), and the second is the start of the fully qualified HFS file name.

You can also specify relative HFS file names in place of fully qualified names. See the following example:

```
open file(Payroll) title('./June.Dat,append(n),recsize(52)');
```

The dataset name `June.Dat` will be prefixed with the pathname of the current z/OS UNIX directory.

With this form, PL/I obtains all DD information either from the TITLE expression or from the ENVIRONMENT attribute of a file declaration - a DD_DDNAME environment variable is not referenced.

The following examples show how to use the OPEN statement with PL/I dynamic allocation support:

- For a z/OS data set:

  ```
  OPEN FILE(QSAM01) title('DSN(USER.FILE.EXT),SHR');
  ```

- For an HFS file:

  ```
  OPEN FILE(QSAM01) title('PATH(/u/USER/sample.txt)');
  ```

Attempting to use files not associated with data sets

If you attempt to use a file that has not been associated with a data set, (either through the use of the TITLE option of the OPEN statement or by establishing a DD_DDNAME environment variable), the UNDEFINEDFILE condition is raised.

The only exceptions are the files SYSIN and SYSPRINT; these default to stdin and stdout, respectively.

How PL/I finds data sets

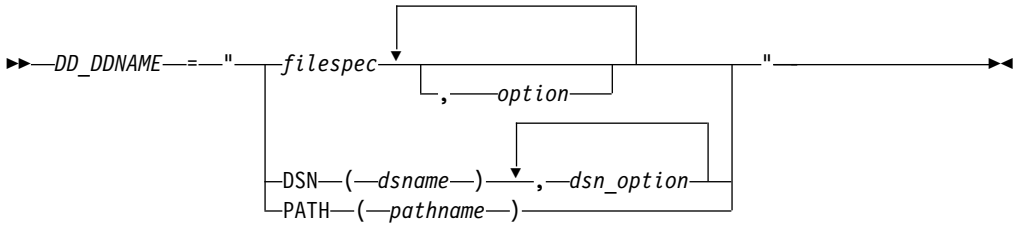
PL/I establishes the path for creating new data sets or accessing existing data sets in one of the following ways:

- The current directory
- The paths as defined by the *export DD_DDNAME* environment variable

Specifying characteristics using DD_DDNAME environment variables

You use the `export` command to establish an environment variable that identifies the data set to be associated with a PL/I file, and, optionally, provide additional characteristics of that data set. This information provided by the environment variable is called data definition (or DD) information.

DD_DDNAME syntax



Blanks are acceptable within the syntax. In addition, the syntax of the statement is not checked at the time the command is entered. It is verified when the data set is opened. If the syntax is wrong, UNDEFINEDFILE is raised with the oncode 96.

DD_DDNAME
The name of the environment variable

The DDNAME must be in uppercase and can be either the name of a file constant or an alternate DDNAME that you specify in the TITLE option of your OPEN statement. For more information, see "Using the TITLE option of the OPEN statement" on page 213.

If you use an alternate DDNAME that is longer than 31 characters, only the first 31 characters are used in forming the environment variable name.

filespec
The specifications of a file or the name of a device to be associated with the PL/I file

option
The options that you can specify as DD information

dsname
The fully-qualified MVS data set name

pathname
The fully-qualified HFS path name

The following topics describe the options that you can specify as DD information. Note that these options do no apply to the DSN() or PATH() formats.

APPEND

The APPEND option specifies whether an existing data set is to be extended or re-created.

Y Specifies that new records are to be added to the end of a sequential data set, or inserted in a relative or indexed data set.

N Specifies that, if the file exists, it is to be re-created.

The APPEND option applies only to OUTPUT files. APPEND is ignored under the following conditions:
- The file does not exist.
- The file does not have the OUTPUT attribute.
- The organization is REGIONAL(1).

BUFSIZE

The BUFSIZE option specifies the number of bytes for a buffer.

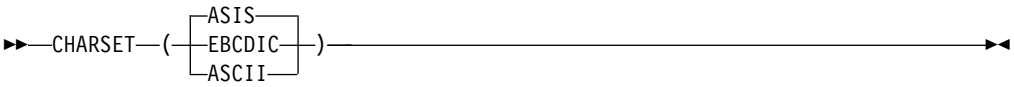
RECORD output is buffered by default and has a default value for BUFSIZE of 64k. STREAM output is buffered, but not by default; the default BUFSIZE value for STREAM output is zero.

If the value of BUFSIZE is zero, the number of bytes for buffering is equal to the value specified in the RECSIZE or LRECL option.

The BUFSIZE option is valid only for a consecutive binary file. If the file is used for terminal input, you should assign the value of zero to BUFSIZE for increased efficiency.

CHARSET for record I/O

This version of the CHARSET option applies only to consecutive files using record I/O. It allows the user to use ASCII data files as input files and specify the character set of output files.



Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file to have (output).

CHARSET for stream I/O

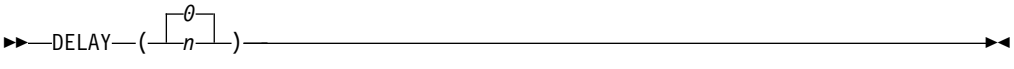
This version of the CHARSET option applies for stream input and output files. It allows the user to use ASCII data files as input files and specify the character set of output files.

If you attempt to specify ASIS when using stream I/O, no error is issued and character sets are treated as EBCDIC.

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file to have (output).

DELAY

The DELAY option specifies the number of milliseconds to delay before the compiler retries an operation that fails when a file or record lock cannot be obtained by the system.

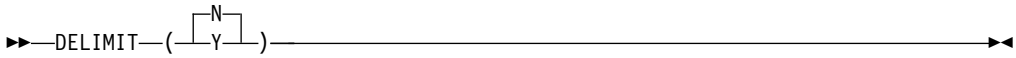


This option is applicable only to VSAM files.

DELIMIT

The DELIMIT option specifies whether the input file contains field delimiters.

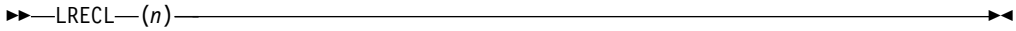
A field delimiter is a blank or a user-defined character that separates the fields in a record. This is applicable for sort input files only.



The sort utility distinguishes text files from binary files with the presence of field delimiters. Input files that contain field delimiters are processed as text files; otherwise, they are considered to be binary files. The library needs this information in order to pass the correct parameters to the sort utility.

LRECL

The LRECL option is the same as the RECSIZE option.



If LRECL is not specified and not implied by a LINESIZE value (except for TYPE(FIXED) files, the default is 1024.

LRMSKIP

The LRMSKIP option allows output to begin on the *n*th line of the first page for the first SKIP format item to be executed after a file is opened. *n* refers to the value specified with the SKIP option of the PUT or GET statement.

If *n* is zero or 1, output begins on the first line of the first page.

PROMPT

The PROMPT option specifies whether colons should be visible as prompts for stream input from the terminal.

PUTPAGE

The PUTPAGE option specifies whether the form feed character should be followed by a carriage return character. This option applies only to printer-destined files.

Printer-destined files are stream output files declared with the PRINT attribute, or record output files declared with the CTLASA environment option.

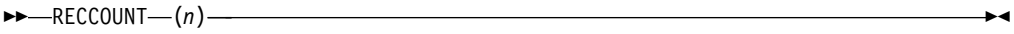
NOCR
Indicates that the form feed character ('0C'x) is not followed by a carriage return character ('0D'x).

CR Indicates that the carriage return character is appended to the form feed character. This option should be specified if output is sent to non-IBM printers.

RECCOUNT

The RECCOUNT option specifies the maximum number of records that can be loaded into a relative or regional data set that is created during the PL/I file opening process.

Note: The RECCOUNT option is not supported for 64-bit programs in PL/I V5.1.



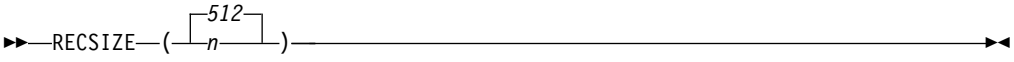
The RECCOUNT option is ignored if PL/I does not create or re-create the data set.

The default for the RECCOUNT option is 50.

Note: Under z/OS, it is recommended that you omit the TITLE option with both the /filespec parameter and RECCOUNT parameter for improved functionality and performance of REGIONAL(1) data sets. In such a case, the number of records that will be loaded into the file depends on the space allocated to the first extent of the data set. See Chapter 13, “Defining and using regional data sets,” on page 279 for additional information.

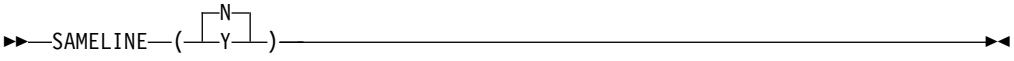
RECSIZE

The RECSIZE option specifies the length of records in the data set. For regional and fixed-length data sets, RECSIZE specifies the length of each record in the data set; for all other data set types, RECSIZE specifies the maximum length that records can have.



SAMELINE

The SAMELINE option specifies whether the system prompt occurs on the same line as the statement that prompts for input.



The following examples show the results of certain combinations of the PROMPT and SAMELINE options.

Example 1

Given the statement `PUT SKIP LIST('ENTER:');`, output is as follows:

```
prompt(y) sameline(y)       ENTER: (cursor)
```

prompt(y) sameline(n)	ENTER: (cursor)
prompt(n) sameline(y)	ENTER: (cursor)
prompt(n) sameline(n)	ENTER: (cursor)

Example 2

Given the statement `PUT SKIP LIST('ENTER');`, output is as follows:

prompt(y) sameline(y)	ENTER: (cursor)
prompt(y) sameline(n)	ENTER : (cursor)
prompt(n) sameline(y)	ENTER (cursor)
prompt(n) sameline(n)	ENTER (cursor)

SKIP0

The SKIP0 option specifies where the line cursor moves when SKIP(0) statement is coded in the source program. SKIP0 applies to terminal files that are not linked as PM applications.

SKIP0(N)
Specifies that the cursor moves to the beginning of the next line.

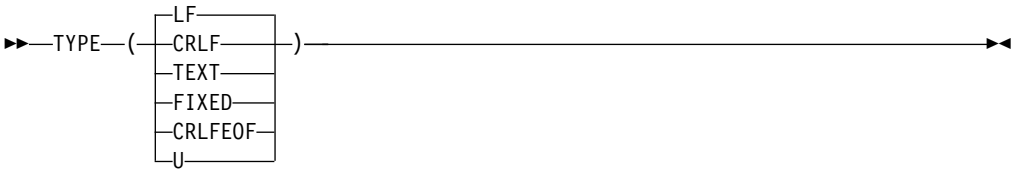
SKIP0(Y)
Specifies that the cursor moves to the beginning of the current line.

The following example shows how you could make the output to the terminal skip zero lines so that the cursor moves to the beginning of the current output line:

```
export DD_SYSPRINT='stdout:,SKIP0(Y)'
```

TYPE

The TYPE option specifies the format of records in a native file.



CRLF
Specifies that records are delimited by the CR - LF character combination. ('CR' and 'LF' represent the ASCII values of carriage return and line feed, '0D'x and '0A'x, respectively. For an output file, PL/I places the characters at the end of each record; for an input file, PL/I discards the characters. For both input and output, the characters are not counted in consideration for RECSIZE.

The data set must not contain any record that is longer than the value determined for the record length of the data set.

LF Specifies that records are delimited by the LF character combination. ('LF' represents the ASCII values of feed or '0A'x.) For an output file, PL/I places the characters at the end of each record; for an input file, PL/I discards the characters. For both input and output, the characters are not counted in consideration for RECSIZE.

The data set must not contain any record that is longer than the value determined for the record length of the data set.

TEXT
Equivalent to LF.

FIXED
Specifies that each record in the data set has the same length. The length determined for records in the data set is used to recognize record boundaries.

All characters in a TYPE(FIXED) file are considered as data, including control characters if they exist. Make sure the record length you specify reflects the presence of these characters, or make sure the record length you specify accounts for all characters in the record.

CRLFEOF
Except for output files, this suboption specifies the same information as CRLF. When one of these files is closed for output, an end-of-file marker is appended to the last record.

U Indicates that records are unformatted. These unformatted files cannot be used by any record or stream I/O statements except OPEN and CLOSE. You can read from a TYPE(U) file only by using the FILEREAD built-in function. You can write to a TYPE(U) file only by using the FILEWRITE built-in function.

The TYPE option applies only to CONSECUTIVE files, except that it is ignored for printer-destined files with ASA(N) applied.

If your program attempts to access an existing data set with TYPE(FIXED) in effect and the length of the data set is not a multiple of the logical record length you specify, PL/I raises the UNDEFINEDFILE condition.

When nonprint files with the TYPE(FIXED) attribute are used, SKIP is replaced by trailing blanks to the end of the line. If TYPE(LF) is being used, SKIP is replaced by LF with no trailing blanks.

Establishing data set characteristics

A data set consists of records stored in a particular format that the operating system data management routines understand. When you declare or open a file in your program, you are describing to PL/I and to the operating system the characteristics of the records that file will contain.

You can also use JCL or an expression in the TITLE option of the OPEN statement to describe to the operating system the characteristics of the data in data sets or in the PL/I files associated with the data sets.

You do not always need to describe your data both within the program and outside it; often one description will serve for both data sets and their associated PL/I files. There are, in fact, advantages to describing the characteristics of your data in only one place.

To effectively describe your program data and the data sets you will be using, you need to understand how the operating system moves and stores data.

Blocks and records

The items of data in a data set are arranged in blocks separated by interblock gaps (IBG). (Some manuals refer to these as interrecord gaps.)

A *block* is the unit of data transmitted to and from a data set. Each block contains one record, part of a record, or several records. You can specify the block size in the BLKSIZE parameter of the DD statement or in the BLKSIZE option of the ENVIRONMENT attribute.

A *record* is the unit of data transmitted to and from a program. You can specify the record length in the LRECL parameter of the DD statement, in the TITLE option of the OPEN statement, or in the RECSIZE option of the ENVIRONMENT attribute.

When writing a PL/I program, you need consider only the records that you are reading or writing; but when you describe the data sets that your program will create or access, you must be aware of the relationship between blocks and records.

Blocking conserves storage space in a magnetic storage volume because it reduces the number of interblock gaps, and it can increase efficiency by reducing the number of input/output operations required to process a data set. Records are blocked and deblocked by the data management routines.

Information interchange codes

The normal code in which data is recorded is the Extended Binary Coded Decimal Interchange Code (EBCDIC).

Each character in the ASCII code is represented by a 7-bit pattern and there are 128 such patterns. The ASCII set includes a substitute character (the SUB control character) that is used to represent EBCDIC characters having no valid ASCII code. The ASCII substitute character is translated to the EBCDIC SUB character, which has the bit pattern 00111111.

Record formats

The records in a data set have one of the following formats:

Fixed-length
Variable-length
Undefined-length.

Records can be blocked if required. The operating system will deblock fixed-length and variable-length records, but you must provide code in your program to deblock undefined-length records.

You specify the record format in the RECFM parameter of the DD statement, in the TITLE option of the OPEN statement, or as an option of the ENVIRONMENT attribute.

Fixed-length records

You can specify the following formats for fixed-length records:
F Fixed-length, unblocked
FB Fixed-length, blocked
FS Fixed-length, unblocked, standard
FBS Fixed-length, blocked, standard.

In a data set with fixed-length records, as shown in Figure 25, all records have the same length. If the records are blocked, each block usually contains an equal number of fixed-length records (although a block can be truncated). If the records are unblocked, each record constitutes a block.

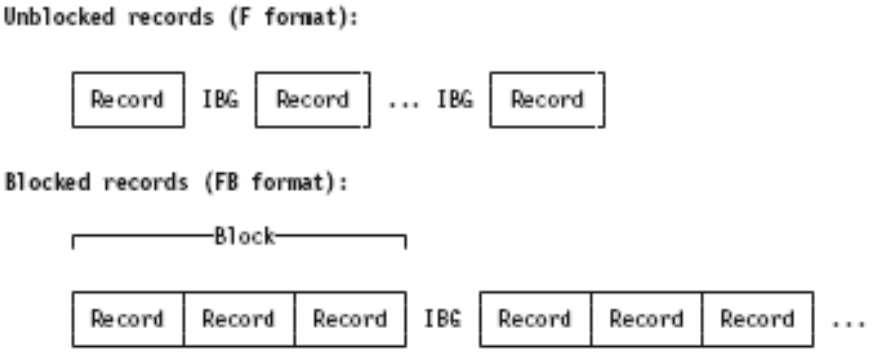


Figure 25. Fixed-length records

Because it bases blocking and deblocking on a constant record length, the operating system processes fixed-length records faster than variable-length records.

Variable-length records

You can specify the following formats for variable-length records:
V Variable-length, unblocked
VB Variable-length, blocked
VS Variable-length, unblocked, spanned
VBS Variable-length, blocked, spanned

V-format allows both variable-length records and variable-length blocks. A 4-byte prefix of each record and the first 4 bytes of each block contain control information for use by the operating system (including the length in bytes of the record or block). Because of these control fields, variable-length records cannot be read backward.

V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record. The first 4 bytes of the block contain block control information, and the next 4 contain record control information.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate. The first 4 bytes of the block contain block control information, and a 4-byte prefix of each record contains record control information.

Spanned Records: A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If this occurs, the record is divided into segments and accommodated in two or more consecutive blocks by specifying the record format as either VS or VBS. Segmentation and assembly are handled by the operating system. The use of spanned records allows you to select a block size, independently of record length, that will combine optimum use of auxiliary storage with maximum efficiency of transmission.

VS-format is similar to V-format. Each block contains only one record or segment of a record. The first 4 bytes of the block contain block control information, and the next 4 contain record or segment control information (including an indication of whether the record is complete or is a first, intermediate, or last segment).

VBS-format differs from VS-format in that each block contains as many complete records or segments as it can accommodate; each block is, therefore, approximately the same size (although there can be a variation of up to 4 bytes, because each segment must contain at least 1 byte of data).

Undefined-length records

U-format allows the processing of records that do not conform to F- and V-formats. The operating system and the compiler treat each block as a record; your program must perform any required blocking or deblocking.

Data set organization

The data management routines of the operating system can handle a number of types of data sets, which differ in the way data is stored within them and in the allowed means of access to the data.

The following table shows the three main types of non-VSAM data sets and the corresponding keywords describing their PL/I organization[1].

Type of data set	PL/I organization
Sequential	CONSECUTIVE or ORGANIZATION(consecutive)
Indexed	INDEXED or ORGANIZATION(indexed)
Direct	REGIONAL or ORGANIZATION(relative)

A fourth type, *partitioned*, has no corresponding PL/I organization.

PL/I also provides support for three types of VSAM data organization: *ESDS*, *KSDS*, and *RRDS*. For more information about VSAM data sets, see Chapter 14, “Defining and using VSAM data sets,” on page 291.

In a *sequential* (or CONSECUTIVE) data set, records are placed in physical sequence. Given one record, the location of the next record is determined by its physical position in the data set. Sequential organization can be selected for direct access devices.

1. Do not confuse the terms *sequential* and *direct* with the PL/I file attributes SEQUENTIAL and DIRECT. The attributes refer to how the file is to be processed, and not to the way the corresponding data set is organized.

An *indexed sequential* (or INDEXED) data set must reside on a direct access volume. An index or a set of indexes maintained by the operating system gives the location of certain principal records. This allows direct retrieval, replacement, addition, and deletion of records, as well as sequential processing.

A *direct* (or REGIONAL) data set must reside on a direct access volume. The data set is divided into regions, each of which contains one or more records. A key that specifies the region number allows direct access to any record; sequential processing is also possible.

In a *partitioned* data set, independent groups of sequentially organized data, each called a member, reside in a direct access data set. The data set includes a directory that lists the location of each member. Partitioned data sets are often called *libraries*. The compiler includes no special facilities for creating and accessing partitioned data sets. Each member can be processed as a CONSECUTIVE data set by a PL/I program. For more information about using partitioned data sets as libraries, see Chapter 9, "Using libraries," on page 239.

Labels

The operating system uses internal labels to identify direct access volumes and to store data set attributes (for example, record length and block size). The attribute information must originally come from a DD statement or from your program.

IBM standard labels have two parts: the initial volume label and header labels. The initial volume label identifies a volume and its owner; the header labels precede and follow each data set on the volume. Header labels contain system information, device-dependent information (for example, recording technique), and dataset characteristics.

Direct access volumes have IBM standard labels. Each volume is identified by a volume label, which is stored on the volume. This label contains a volume serial number and the address of a volume table of contents (VTOC). The table of contents, in turn, contains a label, called a *data set control block* (DSCB), for each data set stored on the volume.

Data Definition (DD) statement

A data definition (DD) statement is a job control statement that defines a data set to the operating system, and is a request to the operating system for the allocation of input/output resources. If the data sets are not dynamically allocated, each job step must include a DD statement for each data set that is processed by the step.

The *z/OS MVS JCL User's Guide* describes the syntax of job control statements. The operand field of the DD statement can contain keyword parameters that describe the location of the data set (for example, volume serial number and identification of the unit on which the volume will be mounted) and the attributes of the data itself (for example, record format).

The DD statement enables you to write PL/I source programs that are independent of the data sets and input/output devices they will use. You can modify the parameters of a data set or process different data sets without recompiling your program.

The LEAVE and REREAD options of the ENVIRONMENT attribute allow you to use the DISP parameter to control the action taken when the end of a

magnetic-tape volume is reached or when a magnetic-tape data set is closed. For information about the LEAVE and REREAD options, see "LEAVE | REREAD" on page 271.

Write validity checking, which was standard in PL/I Version 1, is no longer performed. Write validity checking can be requested through the OPTCD subparameter of the DCB parameter of the JCL DD statement. See the *OS/VS2 Job Control Language* manual.

Use of the conditional subparameters

If you use the conditional subparameters of the DISP parameter for data sets processed by PL/I programs, the step abend facility must be used.

The step abend facility is obtained as follows:

1. The ERROR condition should be raised or signaled whenever the program is to terminate execution after a failure that requires the application of the conditional subparameters.
2. The PL/I user exit must be changed to request an ABEND.

Data set characteristics

The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a data set and the way it will be processed at run time. Whereas the other parameters of the DD statement deal chiefly with the identity, location, and disposal of the data set, the DCB parameter specifies information required for the processing of the records themselves.

The subparameters of the DCB parameter are described in the *OS/VS2 Job Control Language*.

The DCB parameter contains subparameters that describe the following data characteristics:

- The organization of the data set and how it will be accessed (CYLOFL, DSORG, LIMCT, NTM, and OPTCD subparameters)
- Device-dependent information such as the line spacing for a printer (CODE, FUNC, MODE, OPTCD=J, PRTSP, and STACK subparameters)
- The record format (BLKSIZE, KEYLEN, LRECL, and RECFM subparameters)
- The ASA control characters (if any) that will be inserted in the first byte of each record (RECFM subparameter)

You can specify BLKSIZE, LRECL, KEYLEN, and RECFM (or their equivalents) in the ENVIRONMENT attribute of a file declaration in your PL/I program instead of in the DCB parameter.

You cannot use the DCB parameter to override information already established for the data set in your PL/I program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied are ignored.

For a new dataset, the attributes of the file defined in the program will be used if there is a conflict with the DD statement.

You might see message IEC225I with RC=4 issued when closing PDS files. This message can be safely ignored.

The following example of the DCB parameter specifies that fixed-length records, 40 bytes in length, are to be grouped together in a block 400 bytes long:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

Using the TITLE option of the OPEN statement

You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file and, optionally, to provide additional characteristics of the data set.

Related information:

"Using the TITLE option of the OPEN statement" on page 213
You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file, and, optionally, to provide additional characteristics of that data set.

Associating PL/I files with data sets

The execution of a PL/I OPEN statement associates a file with a data set. The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated.

Opening a file

The execution of a PL/I OPEN statement associates a file with a data set. This requires merging of the information describing the file and the data set. If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Subroutines of the PL/I library create a skeleton data control block for the data set. They use the file attributes from the DECLARE and OPEN statements and any attributes implied by the declared attributes, to complete the data control block as far as possible. (See Figure 26 on page 227.) They then issue an OPEN macro instruction, which calls the data management routines to check that the correct volume is mounted and to complete the data control block.

The data management routines examine the data control block to see what information is still needed and then look for this information, first in the DD statement, and finally, if the data set exists and has standard labels, in the data set labels. For new data sets, the data management routines begin to create the labels (if they are required) and to fill them with information from the data control block.

For INPUT data sets, the PL/I program can override the DCB attributes as long as there is no conflict in attributes. For OUTPUT data sets, the PL/I program cannot override the DCB attributes. However, if any DCB attributes are missing from the data set when it is opened, they will be obtained from the PL/I program, if provided.

When the DCB fields are filled in from these sources, control returns to the PL/I library subroutines. If any fields still are not filled in, the PL/I OPEN subroutine provides default information for some of them. For example, if LRECL is not specified, it is provided from the value given for BLKSIZE.

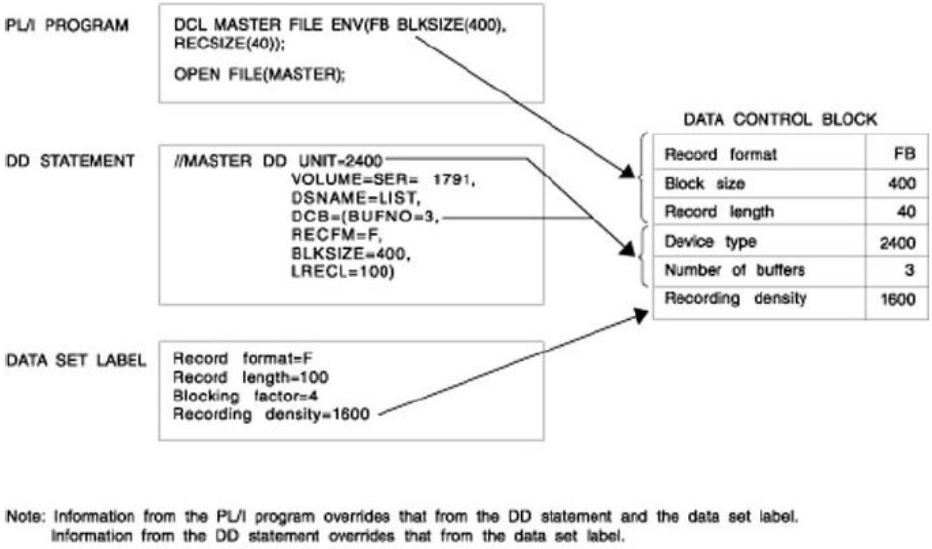


Figure 26. How the operating system completes the DCB

Using system-determined block size:

If you use the system-determined block size function of the Data Facility Product on z/OS, DFP determines the optimal block size for the device type that is assigned.

When you create a new DASD data set, the system derives the optimum block size and saves it in the data set label when all of the following conditions are true:

- Block size is not available or specified from any source. BLKSIZE=0 can be specified.
- You specify LRECL, or it is in the data class. The data set does not have to be SMS managed.
- You specify RECFM, or it is in the data class. It must be fixed or variable.
- You specify DSORG as PS or PO, or if you omit DSORG, it is PS or PO in the data class.

When system-determined block size is active, DFP determines the block size and places it in the data set label before PL/I opens the file. Therefore, if the PL/I program specifies a block size with its ENVIRONMENT option, it must not be in conflict with the value from the system-determined block size.

For detailed information about system-determined block size, see *z/OS DFSMS Using Data Sets*.

Closing a file

The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated.

The PL/I library subroutines first issue a CLOSE macro instruction; when control returns from the data management routines, the subroutines release the data

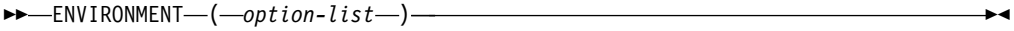
control block that was created when the file was opened. The data management routines complete the writing of labels for new data sets and update the labels of existing data sets.

Specifying characteristics in the ENVIRONMENT attribute

You can use various options in the ENVIRONMENT attribute to specify data characteristics. Each type of file has different attributes and environment options.

The ENVIRONMENT attribute

You use the ENVIRONMENT attribute of a PL/I file declaration file to specify information about the physical organization of the data set associated with a file, and describe other related information. The format of this information must be a parenthesized option list.



Abbreviation: ENV

You can specify the options in any order, separated by blanks or commas.

The following example illustrates the syntax of the ENVIRONMENT attribute in the context of a complete file declaration (the options specified are for VSAM).

```
DCL FILENAME FILE RECORD SEQUENTIAL
    INPUT ENV(VSAM GENKEY);
```

Table 15 summarizes the ENVIRONMENT options and file attributes. Certain qualifications on their use are presented in the notes and comments in the table. NOTE TO REVIEWERS: TO SEE THE TABLE THAT IS TOO COMPLICATED TO USE FOR LIST3270 OUTPUT, PLEASE SEE CHAPTER 5 PAGE 102 OF THE HARDCOPY DOCUMENT.

Table 15. Attributes of PL/I file declarations

Data set type	Stream	Record									Legend:	
		Sequential						Direct				
		Consecutive									C	Checked for VSAM
											D	Default
					Telepro-						I	Must be specified or implied
File Type	Consecutive	Buffered	Unbuffered	Regional	cessing	Indexed	VSAM	Regional	Indexed	VSAM	N	Ignored for VSAM
											O	Optional
											S	Must be specified
											-	Invalid

Table 15. Attributes of PL/I file declarations (continued)

File attributes[1]											Attributes implied
File	I	I	I	I	I	I	I	I	I	I	
Input[1]	D	D	D	D	D	D	D	D	D	D	File
Output	O	O	O	O	O	O	O	O	O	O	File
Environment	I	I	I	S	S	S	S	S	S	S	File
Stream	D	-	-	-	-	-	-	-	-	-	File
Print[1]	O	-	-	-	-	-	-	-	-	-	File stream output
Record	-	I	I	I	I	I	I	I	I	I	File
Update	-	O	O	O	-	O	O	O	O	O	File record
Sequential	-	D	D	D	-	D	D	-	-	D	File record
Buffered	-	D	-	-	I	D	D	-	-	S	File record
Keyed[2]	-	-	-	O	I	O	O	I	I	O	File record
Direct	-	-	-	-	-	-	S	S	S	S	File record keyed

ENVIRONMENT options											Comments
F | FB | FS | FBS | V | VB | VS | VBS | | U	I	S	S	-	-	-	N	-	-	N	VS and VBS are invalid with Stream
F | FB | U	S	S	-	-	-	-	N	-	-	N	ASCII data sets only
F | V | U	-	-	-	S	-	-	N	S	-	N	Only F for REGIONAL(1)
F | FB | V | VB	-	-	-	-	-	S	N	-	S	N	
RECSIZE(n)	I	I	I	I	S	I	C	I	I	C	RECSIZE and/or BLKSIZE must be specified for consecutive
BLKSIZE(n)	I	I	I	I	-	I	N	I	I	N	indexed, and regional files
SCALARVARYING	-	O	O	O	-	O	O	O	O	O	Invalid for ASCII data sets
CONSECUTIVE	D	D	D	-	-	-	O	-	-	O	Allowed for VSAM ESDS
LEAVE | REREAD	O	O	O	-	-	-	-	-	-	-	
CTLASA | CTL360	-	O	O	-	-	-	-	-	-	-	Invalid for ASCII data sets
GRAPHIC	O	-	-	-	-	-	-	-	-	-	
INDEXED	-	-	-	-	-	S	O	-	S	O	Allowed for VSAM ESDS
KEYLOC(n)	-	-	-	-	-	O	-	-	O	-	
ORGANIZATION	D	-	-	-	-	-	-	-	-	-	
GENKEY	-	-	-	-	-	O	O	-	O	O	INPUT or UPDATE files only; KEYED is required
REGIONAL(1)	-	-	-	S	-	-	-	S	-	-	
VSAM	-	-	-	-	-	-	S	-	-	S	
BKWD	-	-	-	-	-	-	O	-	-	O	
REUSE	-	-	-	-	-	-	O	-	-	O	OUTPUT file only

Notes:
1. A file with the INPUT attribute cannot have the PRINT attribute.
2. Keyed is required for INDEXED and REGIONAL output.

Those ENVIRONMENT options that apply to more than one data set organization are described in the topics that follow. In addition, in the following sections, each option is described with each data set organization to which it applies.

Related information:

Chapter 14, “Defining and using VSAM data sets,” on page 291
This chapter covers VSAM (the Virtual Storage Access Method) organization for record-oriented data transmission, VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and the statements you use to load and access the three types of VSAM data sets that PL/I supports—entry-sequenced, key-sequenced, and relative record.

Data set organization options: The following options specify the data set organization:

Each option is described in the discussion of the data set organization to which it applies.

Other ENVIRONMENT options:

You can use a constant or variable with those ENVIRONMENT options that require integer arguments, such as block sizes and record lengths. The variable must not be subscripted or qualified, and must have attributes FIXED BINARY(31,0) and STATIC.

The following list shows ENVIRONMENT options and equivalent DCB parameters.

ENVIRONMENT option	DCB subparameter
Record format	RECFM[1]
RECSIZE	LRECL
BLKSIZE	BLKSIZE
CTLASA|CTL360	RECFM
KEYLENGTH	KEYLEN

Note: [1]VS must be specified as an ENVIRONMENT option, not in the DCB.

Record formats for record-oriented data transmission

Record formats supported depend on the data set organization.

Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
	FS	unblocked, standard
	FBS	blocked, standard
Variable-length	V	unblocked
	VB	blocked
	VS	spanned
	VBS	blocked, spanned
Undefined-length	U	(cannot be blocked)

When U-format records are read into a varying-length string, PL/I sets the length of the string to the block length of the retrieved data.

These record format options do not apply to VSAM data sets. If you specify a record format option for a file associated with a VSAM data set, the option is ignored.

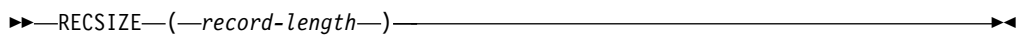
You can specify VS-format records only for data sets with consecutive organization.

Record formats for stream-oriented data transmission

For information about the record format options for stream-oriented data transmission, see "Using stream-oriented data transmission" on page 245.

RECSIZE option

The RECSIZE option specifies the record length.



For files associated with VSAM data sets, *record-length* is the sum of the following:

1. The length required for data

   For variable-length and undefined-length records, this is the maximum length.
2. Any control bytes required

   Variable-length records require 4 (for the record-length prefix); fixed-length and undefined-length records do not require any.

For VSAM data sets, the maximum and average lengths of the records are specified to the Access Method Services utility when the data set is defined. If you include the RECSIZE option in the file declaration for checking purposes, you should specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

You can specify *record-length* as an integer or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:

Fixed-length, and undefined (except ASCII data sets): 32760

V-format, and VS- and VBS-format with UPDATE files: 32756

VS- and VBS-format with INPUT and OUTPUT files: 16777215

ASCII data sets: 9999

VSAM data sets: 32761

Note: For VS- and VBS-format records longer than 32,756 bytes, you must specify the length in the RECSIZE option of ENVIRONMENT, and for the DCB subparameter of the DD statement you must specify LRECL=X. If RECSIZE exceeds the allowed maximum for INPUT or OUTPUT, either a record condition occurs or the record is truncated. UPDATE files are not supported with LRECL=X.

Zero value:
A search for a valid value is made first in the DD statement for the data set associated with the file and second in the data set label:

If neither of these provides a value, the default action is taken (see "Record format, BLKSIZE, and RECSIZE defaults" on page 233).

Negative Value:
The UNDEFINEDFILE condition is raised.

BLKSIZE option

The BLKSIZE option specifies the maximum block size on the data set.

block-size is the sum of the following:

1. The total length(s) of one of the following:
   - A single record
   - A single record and either one or two record segments
   - Several records
   - Several records and either one or two record segments
   - Two record segments
   - A single record segment.

   For variable-length records, the length of each record or record segment includes the 4 control bytes for the record or segment length.

   The preceding list summarizes all the possible combinations of records and record segments options: fixed- or variable-length, blocked or unblocked, spanned or unspanned. When specifying a block size for spanned records, note that each record and each record segment requires 4 control bytes for the record length and that these quantities are in addition to the 4 control bytes required for each block.
2. Any further control bytes required
   - Variable-length blocked records require 4 (for the block size).
   - Fixed-length and undefined-length records do not require any further control bytes.
3. Any block prefix bytes required (ASCII data sets only)

block-size can be specified as an integer, or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:
32760

Zero value:
If you set BLKSIZE to 0, under z/OS the Data Facility Product sets the block size. For more information, see "Record format, BLKSIZE, and RECSIZE defaults."

Negative value:
The UNDEFINEDFILE condition is raised.

The relationship of block size to record length depends on the record format:

FB-format or FBS-format
The block size must be a multiple of the record length.

VB-format:
The block size must be equal to or greater than the sum of the following:
1. The maximum length of any record
2. Four control bytes

VS-format or VBS-format:
The block size can be less than, equal to, or greater than the record length.

Notes:

- Use the BLKSIZE option with unblocked (F- or V-format) records in either of the following ways:
  - Specify the BLKSIZE option, but not the RECSIZE option. Set the record length equal to the block size (minus any control or prefix bytes), and leave the record format unchanged.
  - Specify both BLKSIZE and RECSIZE and ensure that the relationship of the two values is compatible with blocking for the record format you use. Set the record format to FB or VB, whichever is appropriate.
- If for FB-format or FBS-format records the block size equals the record length, the record format is set to F.
- The BLKSIZE option does not apply to VSAM data sets, and is ignored if you specify it for one.

Record format, BLKSIZE, and RECSIZE defaults

If you do not specify either the record format, block size, or record length for a non-VSAM data set, a default action is taken.

Record format:
A search is made in the associated DD statement or data set label. If the search does not provide a value, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets or the foreground terminal, in which case the record format is set to U.

Block size or record length:
If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, and if this value is incompatible with the value in the specified option, the UNDEFINEDFILE condition is raised. If the search is unsuccessful, a value is derived from the specified option (with the addition or subtraction of any control or prefix bytes).

If neither is specified, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets, in which case BLKSIZE is set to 121 for F-format or U-format records and to 129 for V-format records. For files associated with the foreground terminal, RECSIZE is set to 120.

If you are using z/OS with the Data Facility Product system-determined block size, DFP determines the optimum block size for the device type assigned. If you specify BLKSIZE(0) in either the DD assignment or the ENVIRONMENT statement, DFP calculates BLKSIZE by using the record length, record format, and device type.

GENKEY option — key classification

The GENKEY (generic key) option applies only to INDEXED and VSAM key-sequenced data sets. You can use this option to classify keys recorded in a data set and use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

A generic key is a character string that identifies a class of keys; all keys that begin with the string are members of that class. For example, the recorded keys “ABCD”, “ABCE”, and “ABDF” are all members of the classes identified by the generic keys “A” and “AB”, and the first two are also members of the class “ABC”; and the three recorded keys can be considered to be unique members of the classes “ABCD”, “ABCE”, and “ABDF”, respectively.

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an INDEXED data set from the first nondummy record that has a key in a particular class. You identify the class by including its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although you can retrieve the first record that has a key in a particular class by using a READ with the KEY option, you cannot obtain the actual key unless the records have embedded keys, because the KEYTO option cannot be used in the same statement as the KEY option.

In the following example, a key length of more than 3 bytes is assumed:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (GENKEY);
          .
          .
          .
       READ FILE(IND) INTO(INFIELD)
                      KEY ('ABC');
          .
          .
          .
NEXT:  READ FILE (IND) INTO (INFIELD);
          .
          .
          .
       GO TO NEXT;
```

The first READ statement causes the first nondummy record in the data set whose key begins with “ABC” to be read into INFIELD; each time the second READ statement is executed, the nondummy record with the next higher key is retrieved. Repeated execution of the second READ statement results in reading records from higher key classes, because no indication is given when the end of a key class is reached. It is your responsibility to check each key if you do not wish to read

beyond the key class. Any subsequent execution of the first READ statement will reposition the file to the first record of the key class “ABC”.

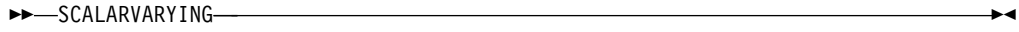
If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

The presence or absence of the GENKEY option affects the execution of a READ statement which supplies a source key that is shorter than the key length specified in the KEYLEN subparameter. This KEYLEN subparameter is found in the DD statement that defines the indexed data set. If you specify the GENKEY option, it causes the source key to be interpreted as a generic key, and the data set is positioned to the first nondummy record in the data set whose key begins with the source key. If you do not specify the GENKEY option, a READ statement's short source key is padded on the right with blanks to the specified key length, and the data set is positioned to the record that has this padded key (if such a record exists). For a WRITE statement, a short source key is always padded with blanks.

Use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered to be the only member of its class).

SCALARVARYING option — varying-length strings

You use the SCALARVARYING option in the input/output of varying-length strings; you can use it with records of any format.



When storage is allocated for a varying-length string, the compiler includes a 2-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if SCALARVARYING is specified for the file.

When you use locate mode statements (LOCATE and READ SET) to create and read a data set with element varying-length strings, you must specify SCALARVARYING to indicate that a length prefix is present, because the pointer that locates the buffer is always assumed to point to the start of the length prefix.

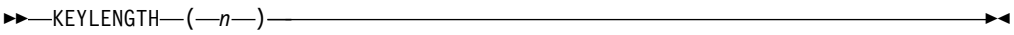
When you specify SCALARVARYING and element varying-length strings are transmitted, you must allow two bytes in the record length to include the length prefix.

A data set created with SCALARVARYING should be accessed only by a file that also specifies SCALARVARYING.

You must not specify SCALARVARYING and CTLASA/CTL360 for the same file, because this causes the first data byte to be ambiguous.

KEYLENGTH option

The KEYLENGTH option specifies the length of the recorded key for KEYED files. You can specify KEYLENGTH for INDEXED files.

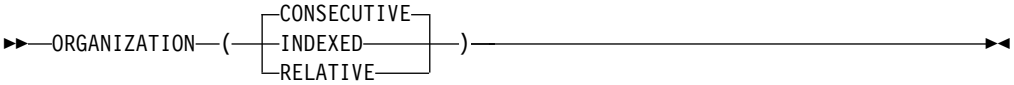


n Specifies the length of the recorded key for KEYED files

If you include the KEYLENGTH option in a VSAM file declaration for checking purposes, and if the key length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

ORGANIZATION option

The ORGANIZATION option specifies the organization of the data set associated with the PL/I file.



CONSECUTIVE
Specifies that the files is associated with a consecutive data set. A consecutive file can be either a native data set or a VSAM, ESDS, RRDS, or KSDS data set.

RELATIVE
Specifies that the file is associated with a relative data set. RELATIVE specifies that the data set contains records that do not have recorded keys. A relative file is a VSAM direct data set. Relative keys range from 1 to nnnn.

Data set types used by PL/I record I/O

Data sets with the RECORD attribute are processed by record-oriented data transmission in which data is transmitted to and from auxiliary storage exactly as it appears in the program variables; no data conversion takes place. A record in a data set corresponds to a variable in the program.

Table 16 shows the facilities that are available with the various types of data sets that can be used with PL/I record I/O.

Table 16. A comparison of data set types available to PL/I record I/O

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
SEQUENCE	Key order	Entry order	Num-bered	Key order	Entry order	By region
DEVICES	DASD	DASD	DASD	DASD	DASD, card, etc.	DASD
ACCESS						
1 By key **2** Sequential **3** Backward	123	123	123	12	2	12
Alternate index access as above	123	123	No	No	No	No
How extended	With new keys	At end	In empty slots	With new keys	At end	In empty slots

Table 16. A comparison of data set types available to PL/I record I/O (continued)

		VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
DELETION							
		Yes, 1	No	Yes, 1	Yes, 2	No	Yes, 1
1	Space reusable						
2	Space not reusable						

The following sections describe how to use Record I/O data sets for different types of data sets:

- Chapter 10, "Defining and using consecutive data sets," on page 245
- Chapter 13, "Defining and using regional data sets," on page 279
- Chapter 14, "Defining and using VSAM data sets," on page 291

Setting environment variables under z/OS UNIX

Some environment variables can be set and exported for use with z/OS UNIX. To set the environment variables system wide so all users have access to them, add the lines suggested in the subsections to the file `/etc/profile`. To set them for a specific user only, add them to the file `.profile` in the user's home directory.

The variables are set the next time the user logs on.

The following example illustrates how to set environment variables:

```
LANG=ja_JP
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
LIBPATH=/home/joe/usr/lib:/home/joe/mylib:/usr/lib
export LANG NLSPATH LIBPATH
```

Rather than using the last statement in the previous example, you can add `export` to each of the preceding lines (`export LANG=ja_JP...`).

You can use the ECHO command to determine the current setting of an environment variable. To define the value of BYPASS, you can use either of the following two examples:

```
echo $LANG

echo $LIBPATH
```

PL/I standard files (SYSPRINT and SYSIN) under z/OS UNIX

SYSIN is read from `stdin` and SYSPRINT is directed to `stdout` by default. If you want either to be associated differently, you must use the TITLE option of the OPEN statement, or establish a DD_DDNAME environment variable naming a data set or another device.

Related information:

"Setting environment variables under z/OS UNIX"
Some environment variables can be set and exported for use with z/OS UNIX. To set the environment variables system wide so all users have access to them, add the lines suggested in the subsections to the file `/etc/profile`. To set them for a specific user only, add them to the file `.profile` in the user's home directory.

Redirecting standard input, output, and error devices under z/OS UNIX

You can redirect standard input, standard output, and standard error devices to a file.

For example, you can use redirection in the following program:

```
Hello2: proc options(main);
  put list('Hello!');
end;
```

After compiling and linking the program, you can invoke it from the command line by entering the following command:

```
hello2 > hello2.out
```

If you want to combine `stdout` and `stderr` in a single file, enter the following command:

```
hello2 > hello2.out 2>&1
```

As is true with display statements, the *greater than* sign redirects the output to the file that is specified after it, in this case `hello2.out`. This means that the word 'Hello' is written in the file `hello2.out`. Note also that the output includes printer control characters because the PRINT attribute is applied to SYSPRINT by default.

READ statements can also access data from `stdin`.

Chapter 9. Using libraries

Within the z/OS operating system, the terms *partitioned data set*, *partitioned data set/extension*, and *library* are synonymous and refer to a type of data set that can be used for the storage of other data sets (usually programs in the form of source, object, or load modules).

A library must be stored on direct access storage and be wholly contained in one volume. It contains independent, consecutively organized data sets, called members. Each member has a unique name, not more than 8 characters long, which is stored in a directory that is part of the library. All the members of one library must have the same data characteristics because only one data set label is maintained.

You can create members individually until there is insufficient space left for a new entry in the directory, or until there is insufficient space for the member itself. You can access members individually by specifying the member name.

Use DD statements or their conversational mode equivalent to create and access members.

You can delete members by using the IBM utility program IEHPROGM. This program deletes the member name from the directory so that the member can no longer be accessed, but you cannot use the space occupied by the member itself again unless you re-create the library or compress the unused space by using, for example, the IBM utility program IEBCOPY. If you attempt to delete a member by using the DISP parameter of a DD statement, it causes the whole data set to be deleted.

Types of libraries

Types of libraries include the system program library, the system procedure library, and private program libraries.

You can use the following types of libraries with a PL/I program:

- The system program library SYS1.LINKLIB or its equivalent

  This library can contain all system processing programs such as compilers and the linkage editor.
- Private program libraries

  These libraries usually contain user-written programs. It is often convenient to create a temporary private library to store the load module output from the linkage editor until it is executed by a later job step in the same job. The temporary library will be deleted at the end of the job. Private libraries are also used for automatic library call by the linkage editor and the loader.
- The system procedure library SYS1.PROCLIB or its equivalent

  This library contains the job control procedures that have been cataloged for your installation.

Using a library

A PL/I program can use a library directly.

© Copyright IBM Corp. 1999, 2017

If you are adding a new member to a library, its directory entry will be made by the operating system when the associated file is closed, using the member name specified as part of the data set name.

If you are accessing a member of a library, its directory entry can be found by the operating system from the member name that you specify as part of the data set name.

More than one member of the same library can be processed by the same PL/I program, but only one such file can be open as output at any one time. You access different members by giving the member name in a DD statement.

Creating a library

To create a library, include in your job step a DD statement containing the information required for creating a library.

See Table 17 for the information required when you create a library. The information required is similar to that for a consecutively organized data set (see "Defining files using record I/O" on page 268) except for the SPACE parameter.

Table 17. Information required when you create a library

Information required	Parameter of DD statement
Type of device that will be used	UNIT=
Serial number of the volume that will contain the library	VOLUME=SER
Name of the library	DSNAME=
Amount of space required for the library	SPACE=
Disposition of the library	DISP=

SPACE parameter

You can use the SPACE parameter in a DD statement to specify the amount of space required for the library that you want to create.

The SPACE parameter in a DD statement that defines a library must always be of the form:

```
SPACE=(units,(quantity,increment,directory))
```

The third term (increment) is optional, and you can indicate its absence by a comma. The last term, which specifies the number of directory blocks to be allocated, is required.

The amount of auxiliary storage required for a library depends on the number and sizes of the members to be stored in it and on how often members will be added or replaced. (Space occupied by deleted members is not released.) The number of directory blocks required depends on the number of members and the number of aliases. You can specify an incremental quantity in the SPACE parameter that allows the operating system to obtain more space for the data set, if such is necessary at the time of creation or at the time a new member is added; the number of directory blocks, however, is fixed at the time of creation and cannot be increased.

Example

In the following example, the DD statement requests the job scheduler to allocate 5 cylinders of the DASD with a volume serial number 3412 for a new library name ALIB and to enter this name in the system catalog. The last term of the SPACE parameter requests that part of the space allocated to the data set be reserved for ten directory blocks.

```
//   PDS DD UNIT=SYSDA,VOL=SER=3412,
//   DSNAME=ALIB,
//   SPACE=(CYL,(5,,10)),
//   DISP=(,CATLG)
```

Creating and updating a library member

When you create and update library members, you must follow the guidelines in this topic.

The members of a library must have identical characteristics. Otherwise, you might later have difficulty retrieving them. Identical characteristics are necessary because the volume table of contents (VTOC) will contain only one data set control block (DSCB) for the library and not one for each member. When you use a PL/I program to create a member, the operating system creates the directory entry; you cannot place information in the user data field.

When you create a library and a member at the same time, your DD statement must include all the parameters listed under "Creating a library" on page 240 (although you can omit the DISP parameter if the data set is to be temporary). The DSNAME parameter must include the member name in parentheses. For example, DSNAME=ALIB(MEM1) names the member MEM1 in the data set ALIB. If the member is placed in the library by the linkage editor, you can use the linkage editor NAME statement or the NAME compile-time option instead of including the member name in the DSNAME parameter. You must also describe the characteristics of the member (such as, record format, and so on) either in the DCB parameter or in your PL/I program. These characteristics will also apply to other members added to the data set.

When creating a member to be added to an existing library, you do not need the SPACE parameter. The original space allocation applies to the whole of the library and not to an individual member. Furthermore, you do not need to describe the characteristics of the member, because these are already recorded in the DSCB for the library.

To add two more members to a library in one job step, you must include a DD statement for each member, and you must close one file that refers to the library before you open another.

Example: Creating new libraries for compiled object modules

This example uses the cataloged procedure IBMZC to compile a simple PL/I program and place the object module in a new library named EXLIB. The DD statement that defines the new library and names the object module overrides the DD statement SYSLIN in the cataloged procedure.

The PL/I program is a function procedure that, given two values in the form of the character string produced by the TIME built-in function, returns the difference in milliseconds.

```
//OPT10#1 JOB
//TR        EXEC  IBMZC
//PLI.SYSLIN  DD  UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//      SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSIN   DD  *
   ELAPSE:  PROC(TIME1,TIME2);
     DCL (TIME1,TIME2) CHAR(9),
         H1 PIC '99' DEF TIME1,
         M1 PIC '99' DEF TIME1 POS(3),
         MS1 PIC '99999' DEF TIME1 POS(5),
         H2 PIC '99' DEF TIME2,
         M2 PIC '99' DEF TIME2 POS(3),
         MS2 PIC '99999' DEF TIME2 POS(5),
         ETIME FIXED DEC(7);
     IF H2<H1 THEN H2=H2+24;
     ETIME=((H2*60+M2)*60000+MS2)-((H1*60+M1)*60000+MS1);
     RETURN(ETIME);
   END ELAPSE;
/*
```

Figure 27. Creating new libraries for compiled object modules

Example: Placing a load module in an existing library

The example uses the cataloged procedure IBMZCL to compile and link-edit a PL/I program and place the load module in the existing library HPU8.CCLM.

```
//OPT10#2  JOB
//TRLE      EXEC  IBMZCL
//PLI.SYSIN   DD  *
   MNAME:  PROC  OPTIONS(MAIN);
     .
     .
     .
     program
     .
     .
     .

   END MNAME;
/*
//LKED.SYSLMOD  DD  DSNAME=HPU8.CCLM(DIRLIST),DISP=OLD
```

Figure 28. Placing a load module in an existing library

Example: Updating a library member

To use a PL/I program to add or delete one or more records within a member of a library, you must rewrite the entire member in another part of the library. This is rarely an economic proposition because the space originally occupied by the member cannot be used again. You must use two files in your PL/I program, but both can be associated with the same DD statement.

The program shown in Figure 30 on page 243 updates the member created by the program in Figure 29 on page 243. It copies all the records of the original member

except those that contain only blanks.

```
//OPT10#3  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  NMEM:  PROC OPTIONS(MAIN);
     DCL IN FILE RECORD SEQUENTIAL INPUT,
         OUT FILE RECORD SEQUENTIAL OUTPUT,
         P POINTER,
         IOFIELD CHAR(80) BASED(P),
         EOF BIT(1) INIT('0'B);
     OPEN FILE(IN),FILE (OUT);
     ON ENDFILE(IN) EOF='1'B;
     READ FILE(IN) SET(P);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
     WRITE FILE(OUT) FROM(IOFIELD);
     READ FILE(IN) SET(P);
     END;
     CLOSE FILE(IN),FILE(OUT);
  END NMEM;
/*
//GO.OUT   DD  UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//     DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//     DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
  MEM:  PROC OPTIONS(MAIN);
        /* this is an incomplete dummy library member */
```

Figure 29. Creating a library member in a PL/I program

```
//OPT10#4  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  UPDTM: PROC OPTIONS(MAIN);
     DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
         EOF BIT(1) INIT('0'B),
         DATA CHAR(80);
     ON ENDFILE(OLD)  EOF = '1'B;
     OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');
     READ FILE(OLD) INTO(DATA);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
     IF DATA=' ' THEN ;
     ELSE WRITE FILE(NEW) FROM(DATA);
     READ FILE(OLD) INTO(DATA);
     END;
   CLOSE FILE(OLD),FILE(NEW);
  END UPDTM;
/*
//GO.OLD DD  DSNAME=HPU8.ALIB(NMEM),DISP=(OLD,KEEP)
```

Figure 30. Updating a library member

Extracting information from a library directory

The directory of a library is a series of records (entries) at the beginning of the data set. There is at least one directory entry for each member. Each entry contains a member name, the relative address of the member within the library, and a variable amount of user data.

User data is information inserted by the program that created the member. An entry that refers to a member (load module) written by the linkage editor includes user data in a standard format, described in the systems manuals.

If you use a PL/I program to create a member, the operating system creates the directory entry for you and you cannot write any user data. However, you can use assembler language macro instructions to create a member and write your own user data. The method for using macro instructions to do this is described in the data management manuals.

Chapter 10. Defining and using consecutive data sets

This chapter covers consecutive data set organization and the ENVIRONMENT options that define consecutive data sets for stream and record-oriented data transmission. It then covers how to create, access, and update consecutive data sets for each type of transmission.

In a data set with consecutive organization, records are organized solely on the basis of their successive physical positions; when the data set is created, records are written consecutively in the order in which they are presented. You can retrieve the records only in the order in which they were written. See Table 15 on page 228 for valid file attributes and ENVIRONMENT options for consecutive data sets.

Using stream-oriented data transmission

This section covers how to define data sets for use with PL/I files that have the STREAM attribute. It covers the ENVIRONMENT options you can use and how to create and access data sets. The essential parameters of the DD statements you use in creating and accessing these data sets are summarized in tables, and several examples of PL/I programs are included to illustrate the text.

Data sets with the STREAM attribute are processed by stream-oriented data transmission, which allows your PL/I program to ignore block and record boundaries and treat a data set as a continuous stream of data values in character or graphic form.

You create and access data sets for stream-oriented data transmission by using the list-, data-, and edit-directed input and output statements described in the *PL/I Language Reference*.

For output, PL/I converts the data items from program variables into character form if necessary, and builds the stream of characters or graphics into records for transmission to the data set.

For input, PL/I takes records from the data set and separates them into the data items requested by your program, converting them into the appropriate form for assignment to program variables.

You can use stream-oriented data transmission to read or write graphic data. There are terminals, printers, and data-entry devices that, with the appropriate programming support, can display, print, and enter graphics. You must be sure that your data is in a format acceptable for the intended device or for a print utility program.

Defining files using stream I/O

You can define files for stream-oriented data transmission by a file declaration.

```
DCL filename FILE STREAM
             INPUT | {OUTPUT [PRINT]}
             ENVIRONMENT(options);
```

For information about the default file attributes, see Table 15 on page 228. For information about the FILE attribute, see the *PL/I Language Reference*. For more

© Copyright IBM Corp. 1999, 2017

information about the PRINT attribute, see "Using PRINT files with stream I/O" on page 254. For information about the options of the ENVIRONMENT attribute, see "Specifying ENVIRONMENT options."

Defining stream files using PL/I dynamic allocation

To define the stream files, you can use a DD statement, an environment variable, or the TITLE option of the OPEN statement.

When an environment variable or the TITLE option is used, the name must be in uppercase. Specify the MVS data set in one of the following ways:

- DSN(*data-set-name*)
- DSN(*data-set-name*(*member-name*))

  data-set-name must be fully qualified and cannot be a temporary data set; for example, it must not start with &.

Specify the HFS file as follows:

```
PATH(absolute-path-name)
```

You can specify the following attributes in any order after the DSN keyword:

```
NEW, OLD, SHR, or MOD
TRACKS or CYL
SPACE(n,m)
VOL(volser)
UNIT(type)
KEEP, DELETE, CATALOG, or UNCATALOG
STORCLAS(storageclass)
MGMTCLAS(managementclass)
DATACLAS(dataclass)
```

Note: You cannot create a PDS or PDSE by using an environment variable or the TITLE option of the OPEN statement, but you can create a new member in an existing PDS or PDSE.

Specifying ENVIRONMENT options

This topic describes the ENVIRONMENT options that define consecutive data sets for stream and record-oriented data transmission.

Table 15 on page 228 summarizes the ENVIRONMENT options. The following options are applicable to stream-oriented data transmission:

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
F|FB|FS|FBS|V|VB|VS|VBS|U
RECSIZE(record-length)
BLKSIZE(block-size)
GRAPHIC
LEAVE|REREAD
```

For information about how to specify these options for stream-oriented data transmission, see the following topics.

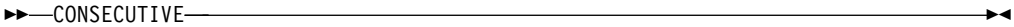
Option	See this topic
CONSECUTIVE	"CONSECUTIVE" on page 247
F | FB | FS | FBS | V | VB | VS | VBS | U	"Record format options" on page 247
RECSIZE	"RECSIZE" on page 248
BLKSIZE	"BLKSIZE option" on page 232

Option	See this topic
GRAPHIC	"GRAPHIC" on page 248
LEAVE|REREAD	"LEAVE|REREAD" on page 271

CONSECUTIVE

STREAM files must have CONSECUTIVE data set organization; however, it is not necessary to specify this in the ENVIRONMENT options because CONSECUTIVE is the default data set organization.

The CONSECUTIVE option for STREAM files is the same as that described in "Data set organization" on page 223.



Record format options

Although record boundaries are ignored in stream-oriented data transmission, record format is important when you create a data set. This is not only because record format affects the amount of storage space occupied by the data set and the efficiency of the program that processes the data, but also because the data set can later be processed by record-oriented data transmission.

Having specified the record format, you need not concern yourself with records and blocks as long as you use stream-oriented data transmission. You can consider your data set a series of characters or graphics arranged in lines, and you can use the SKIP option or format item (and, for a PRINT file, the PAGE and LINE options and format items) to select a new line.

Records can have one of the following formats, which are described in details in "Record formats" on page 221.

Fixed-length	F	unblocked
	FB	blocked
	FS	unblocked, standard
	FBS	blocked, standard
Variable-length	V	unblocked
	VB	blocked
	VS	
	VBS	
Undefined-length	U	(cannot be blocked)

Blocking and deblocking of records are performed automatically.

RECSIZE

RECSIZE for stream-oriented data transmission is the same as that described in "Specifying characteristics in the ENVIRONMENT attribute" on page 228. Additionally, a value specified by the LINESIZE option of the OPEN statement overrides a value specified in the RECSIZE option. LINESIZE is discussed in the *PL/I Language Reference*.

Additional record-size considerations for list- and data-directed transmission of graphics are given in the *PL/I Language Reference*.

Defaults for record format, BLKSIZE, and RECSIZE

If you do not specify the record format, BLKSIZE, or RECSIZE option in the ENVIRONMENT attribute, or in the associated DD statement or data set label, PL/I determines the default values.

- Input files:

  Defaults are applied as for record-oriented data transmission, described in "Record format, BLKSIZE, and RECSIZE defaults" on page 233.

- Output files:

  Record format
  Set to VB-format.

  Record length
  The specified or default LINESIZE value is used:
  – PRINT files:
  - F, FB, FBS, or U: line size + 1
  - V or VB: line size + 5
  – Non-PRINT files:
  - F, FB, FBS, or U: linesize
  - V or VB: linesize + 4

  Block Size
  Files associated with SYSOUT:
  – F, FB, or FBS: record length
  – V or VB: record length + 4

  New or temporary data set:
  – Optimum block size determined by DFP

GRAPHIC

Specify the GRAPHIC option for edit-directed I/O.

The ERROR condition is raised for list- and data-directed I/O if you have graphics in input or output data and do not specify the GRAPHIC option.

For edit-directed I/O, the GRAPHIC option specifies that left and right delimiters are added to DBCS variables and constants on output, and that input graphics will have left and right delimiters. If you do not specify the GRAPHIC option, left and right delimiters are not added to output data, and input graphics do not require left and right delimiters. When you do specify the GRAPHIC option, the ERROR condition is raised if left and right delimiters are missing from the input data.

For information about the graphic data type and about the G-format item for edit-directed I/O, see the *PL/I Language Reference*.

Creating a data set with stream I/O

To create a data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set.

For z/OS UNIX, use one of the following methods to provide the additional information:
- TITLE option of the OPEN statement
- DD_DDNAME environment variable
- ENVIRONMENT attribute

The following topics describe the essential information that you need to provide to create a data set and discuss some of the optional information you can supply.

Essential information

When your application creates a STREAM file, PL/I will derive a line-size value for that file.

PL/I derives the line-size value from one of the following sources in order of declining precedence.
- LINESIZE option of the OPEN statement
- RECSIZE option of the ENVIRONMENT attribute
- RECSIZE option of the TITLE option of the OPEN statement
- RECSIZE option of the DD_DDNAME environment variable
- PL/I-supplied default value

If a LINESIZE value is supplied but a RECSIZE valus is not, PL/I derives the record-length value as follows:
- For a V-format PRINT file, the value is LINESIZE + 5.
- For a V-format non-PRINT file, the value is LINESIZE + 4.
- For a F-format PRINT file, the value is LINESIZE + 1.
- In all other cases, the value is LINESIZE.

If a LINESIZE value is not supplied but a RECSIZE value is, PL/I derives the line-size value from RECSIZE as follows:
- For a V-format PRINT file, the value is RECSIZE - 5.
- For a V-format non-PRINT file, the value is RECSIZE - 4.
- For a F-format PRINT file, the value is RECSIZE - 1.
- In all other cases, the value is RECSIZE.

If neither LINESIZE nor RECSIZE is supplied, PL/I determines a default line-size value based on the attributes of the file and the type of associated data set. If PL/I cannot supply an appropriate default line size, the UNDEFINEDFILE condition is raised.

A default line-size value is supplied for an OUTPUT file under the following conditions:
- The file has the PRINT attribute. In this case, the value is obtained from the tab control table.

- The associated data set is the terminal (stdout: or stderr:). In this case, the value is 120.

Note that if the LINESIZE option is specified (on the OPEN statement) and RECSIZE is also specified (in the ENVIRONMENT attribute, the TITLE option, or the DD statement), and if the record size value is too small to hold the LINESIZE (taking into account the record format and appropriate control byte overhead), the following occurs:

- If you are using Language Environment for z/OS 1.9 or earlier releases, for DD SYSOUT= files, the LINESIZE option will be used to determine a new record size that matches the given LINESIZE; for DD DSN= files and all other files, the UNDEFINEDFILE condition will be raised.
- If you are using Language Environment for z/OS releases subsequent to 1.9, the UNDEFINEDFILE condition will be raised for all files.

Example: Creating a data set with stream-oriented data transmission

This example shows how to use edit-directed stream-oriented data transmission to create a data set on a direct access storage device.

The data read from the input stream by the file SYSIN includes a field VREC that contains five unnamed 7-character subfields; the field NUM defines the number of these subfields that contain information. The output file WORK transmits to the data set the whole of the field FREC and only those subfields of VREC that contain information.

```
//EX7#2  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM OUTPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 PAD CHAR(25),
               2 VREC CHAR(35),
             EOF BIT(1) INIT('0'B),
             IN CHAR(80) DEF REC;
         ON ENDFILE(SYSIN) EOF='1'B;
         OPEN FILE(WORK) LINESIZE(400);
         GET FILE(SYSIN) EDIT(IN)(A(80));
         DO WHILE (¬EOF);
         PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
         GET FILE(SYSIN) EDIT(IN)(A(80));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(NEW,CATLG),UNIT=SYSDA,
//           SPACE=(TRK,(1,1))
//GO.SYSIN DD *
R.C.ANDERSON       0 202848 DOCTOR
B.F.BENNETT        2 771239 PLUMBER          VICTOR HAZEL
R.E.COLE           5 698635 COOK             ELLEN  VICTOR JOAN   ANN    OTTO
J.F.COOPER         5 418915 LAWYER           FRANK  CAROL  DONALD NORMAN BRENDA
A.J.CORNELL        3 237837 BARBER           ALBERT ERIC   JANET
E.F.FERRIS         4 158636 CARPENTER        GERALD ANNA   MARY   HAROLD
/*
```

Figure 31. Creating a data set with stream-oriented data transmission

Example: Writing graphic data to a stream file

This example shows a program that uses list-directed output to write graphics to a stream file.

This example assumes that you have an output device that can print graphic data. The program reads employee records and selects persons living in a certain area. It then edits the address field, inserting one graphic blank between each address item, and prints the employee number, name, and address.

```
//EX7#3  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN      DD *
% PROCESS GRAPHIC;
  XAMPLE1:  PROC OPTIONS(MAIN);
            DCL  INFILE FILE INPUT RECORD,
                 OUTFILE FILE OUTPUT STREAM ENV(GRAPHIC);
 /* GRAPHIC OPTION MEANS DELIMITERS WILL BE INSERTED ON OUTPUT FILES. */
           DCL
                1 IN,
                   3 EMPNO CHAR(6),
                   3 SHIFT1 CHAR(1),
                   3 NAME,
                      5 LAST G(7),
                      5 FIRST G(7),
                   3 SHIFT2 CHAR(1),
                   3 ADDRESS,
                      5 ZIP CHAR(6),
                      5 SHIFT3 CHAR(1),
                      5 DISTRICT G(5),
                      5 CITY G(5),
                      5 OTHER G(8),
                      5 SHIFT4 CHAR(1);
           DCL EOF BIT(1) INIT('0'B);
           DCL ADDRWK G(20);
     ON ENDFILE (INFILE) EOF = '1'B;
     READ FILE(INFILE) INTO(IN);
     DO WHILE(¬EOF);
             DO;
                 IF SUBSTR(ZIP,1,3)¬='300'
                    THEN LEAVE;
                 L=0;
                 ADDRWK=DISTRICT;
                 DO I=1 TO 5;
                 IF SUBSTR(DISTRICT,I,1)= <   >
                    THEN LEAVE;               /* SUBSTR BIF PICKS 3P  */
                 END;                         /* THE ITH GRAPHIC CHAR */
                 L=L+I+1;                     /* IN DISTRICT          */
                 SUBSTR(ADDRWK,L,5)=CITY;
                 DO I=1 TO 5;
                 IF SUBSTR(CITY,I,1)= <   >
                    THEN LEAVE;
                 END;
                 L=L+I;
                 SUBSTR(ADDRWK,L,8)=OTHER;
                 PUT FILE(OUTFILE) SKIP           /* THIS DATA SET    */
                 EDIT(EMPNO,IN.LAST,FIRST,ADDRWK) /* REQUIRES UTILITY */
                     (A(8),G(7),G(7),X(4),G(20)); /* TO PRINT GRAPHIC */
                                                  /* DATA             */
                 END;                       /* END OF NON-ITERATIVE DO */
         READ FILE(INFILE) INTO (IN);
         END;                                 /* END OF DO WHILE(¬EOF) */
     END XAMPLE1;
 /*
 //GO.OUTFILE   DD  SYSOUT=A,DCB=(RECFM=VB,LRECL=121,BLKSIZE=129)
 //GO.INFILE DD *
 ABCDEF<
>300099<           3 3 3 3 3 3 3   >
 ABCD  <
>300011<           3 3 3 3       >
 /*
```

Figure 32. Writing graphic data to a stream file

Accessing a data set with stream I/O

A data set accessed through stream-oriented data transmission does not require to be created by stream-oriented data transmission, but it must have CONSECUTIVE organization, and all the data in it must be in character or graphic form. You can open the associated file for input, and read the records the data set contains; or you can open the file for output, and extend the data set by adding records at the end.

To access a data set, you must use one of the following ways to identify it:
- ENVIRONMENT attribute
- DD_DDNAME environment variable
- TITLE option of the OPEN statement

The following topics describe the essential information you must include in the DD statement and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

Essential information

When your application accesses an existing STREAM file, PL/I must obtain a record-length value for that file.

If the data set does not have a record-length, the value can come from one of the following sources:
- The LINESIZE option of the OPEN statement
- The RECSIZE option of the ENVIRONMENT attribute
- The RECSIZE option of the DD_DDNAME environment variable
- The RECSIZE option of the TITLE option of the OPEN statement
- PL/I-supplied default value

If you are using an existing OUTPUT file, or if you supply a RECSIZE value, PL/I determines the record-length value as described in "Creating a data set with stream I/O" on page 249.

PL/I uses a default record-length value for an INPUT file under the following conditions:
- The file is SYSIN, value = 80
- The file is associated with the terminal (stdout: or stderr:), value = 120

Record format

When using stream-oriented data transmission to access a data set, you do not need to know the record format of the data set (except when you must specify a block size); each GET statement transfers a discrete number of characters or graphics to your program from the data stream.

If you do give record-format information, it must be compatible with the actual structure of the data set. For example, if a data set is created with F-format records, a record size of 600 bytes, and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes; but if you specify a block size of 3500 bytes, your data will be truncated.

Example: Accessing a data set with stream-oriented data transmission

This example shows a program that accesses a data set with stream-oriented data transmission.

The program in Figure 33 reads the data set created by the program in Figure 31 on page 251 and uses the file SYSPRINT to list the data that it contains.

Each set of data is read, by the GET statement, into two variables: FREC, which always contains 45 characters; and VREC, which always contains 35 characters. At each execution of the GET statement, VREC consists of the number of characters generated by the expression 7*NUM, together with sufficient blanks to bring the total number of characters to 35. The DISP parameter of the DD statement could read simply DISP=OLD; if DELETE is omitted, an existing data set will not be deleted.

```
//EX7#5  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM INPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 SERNO CHAR(7),
                 3 PROF CHAR(18),
               2 VREC CHAR(35),
             IN CHAR(80) DEF REC,
             EOF BIT(1) INIT('0'B);
         ON ENDFILE(WORK) EOF='1'B;
         OPEN FILE(WORK);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)
```

Figure 33. Accessing a data set with stream-oriented data transmission

Using PRINT files with stream I/O

Both the operating system and the PL/I language include features that facilitate the formatting of printed output.

The operating system allows you to use the first byte of each record for a print control character. The control characters, which are not printed, cause the printer to skip to a new line or page. (Tables of print control characters are given in Figure 36 on page 270 and Table 19 on page 271.)

In a PL/I program, the use of a PRINT file provides a convenient means of controlling the layout of printed output from stream-oriented data transmission. The compiler automatically inserts print control characters in response to the PAGE, SKIP, and LINE options and format items.

You can apply the PRINT attribute to any STREAM OUTPUT file, even if you do not intend to print the associated data set directly. When a PRINT file is associated with a direct access data set, the print control characters have no effect on the layout of the data set, but appear as part of the data in the records.

PRINT files opened with FB or VB will cause the UNDEFINEDFILE condition to be raised. PRINT files should be opened with the "A" option, that is, FBA or VBA.

The compiler reserves the first byte of each record transmitted by a PRINT file for an American National Standard print control character, and inserts the appropriate characters automatically.

A PRINT file uses only the following five print control characters:

Character	**Action**
	Space 1 line before printing (blank character)
0	Space 2 lines before printing
-	Space 3 lines before printing
+	No space before printing
1	Start new page

The compiler handles the PAGE, SKIP, and LINE options or format items by padding the remainder of the current record with blanks and inserting the appropriate control character in the next record. If SKIP or LINE specifies more than a 3-line space, the compiler inserts sufficient blank records with appropriate control characters to accomplish the required spacing. In the absence of a print control option or format item, when a record is full, the compiler inserts a blank character (single line space) in the first byte of the next record.

If a PRINT file is being transmitted to a terminal, the PAGE, SKIP, and LINE options will never cause more than 3 lines to be skipped, unless formatted output is specified.

Controlling printed line length

You can limit the length of the printed line produced by a PRINT file either by specifying a record length in your PL/I program (ENVIRONMENT attribute) or in a DD statement, or by giving a line size in an OPEN statement (LINESIZE option).

The record length must include the extra byte for the print control character, that is, it must be 1 byte larger than the length of the printed line (5 bytes larger for V-format records). The value you specify in the LINESIZE option refers to the number of characters in the printed line; the compiler adds the print control character.

The blocking of records has no effect on the appearance of the output produced by a PRINT file, but it does result in more efficient use of auxiliary storage when the file is associated with a data set on a direct access device. If you use the LINESIZE option, ensure that your line size is compatible with your block size. For F-format records, block size must be an exact multiple of (line size+1); for V-format records, block size must be at least 9 bytes greater than line size.

Although you can vary the line size for a PRINT file during execution by closing the file and opening it again with a new line size, you must do so with caution if you are using the PRINT file to create a data set on a direct access device. You cannot change the record format that is established for the data set when the file is first opened. If the line size you specify in an OPEN statement conflicts with the record format already established, the UNDEFINEDFILE condition is raised. To prevent this, either specify V-format records with a block size at least 9 bytes greater than the maximum line size you intend to use, or ensure that the first OPEN statement specifies the maximum line size. (Output destined for the printer

can be stored temporarily on a direct access device, unless you specify a printer by using UNIT=, even if you intend it to be fed directly to the printer.)

Because PRINT files have a default line size of 120 characters, you need not give any record format information for them. In the absence of other information, the compiler assumes V-format records. The complete default information is as follows:

```
BLKSIZE=129
LRECL=125
RECFM=VBA
```

Example

Figure 34 on page 257 illustrates the use of a PRINT file and the printing options of stream-oriented data transmission statements to format a table and write it onto a direct access device for printing on a later occasion. The table comprises the natural sines of the angles from 0° to 359° 54' in steps of 6'.

The statements in the ENDPAGE ON-unit insert a page number at the bottom of each page, and set up the headings for the following page.

The DD statement defining the data set created by this program includes no record-format information. The compiler infers the following from the file declaration and the line size specified in the statement that opens the file TABLE:

Record format =
V (the default for a PRINT file)

Record size =
98 (line size + 1 byte for print control character + 4 bytes for record control field)

Block size =
102 (record length + 4 bytes for block control field)

The program in Figure 38 on page 277 uses record-oriented data transmission to print the table created by the program in Figure 34 on page 257.

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 SINE: PROC OPTIONS(MAIN);
   DCL TABLE       FILE STREAM OUTPUT PRINT;
   DCL DEG         FIXED DEC(5,1) INIT(0);  /* INIT(0) FOR ENDPAGE  */
   DCL MIN         FIXED DEC(3,1);
   DCL PGNO        FIXED DEC(2)   INIT(0);
   DCL ONCODE      BUILTIN;

   ON ERROR
     BEGIN;
       ON ERROR SYSTEM;
       DISPLAY ('ONCODE = '|| ONCODE);
     END;

   ON ENDPAGE(TABLE)
     BEGIN;
       DCL I;
       IF PGNO ¬= 0 THEN
         PUT FILE(TABLE) EDIT ('PAGE',PGNO)
             (LINE(55),COL(80),A,F(3));
       IF DEG ¬= 360 THEN
         DO;
           PUT FILE(TABLE) PAGE EDIT ('NATURAL SINES') (A);
           IF PGNO ¬= 0 THEN
             PUT FILE(TABLE) EDIT ((I DO I = 0 TO 54 BY 6))
                                  (SKIP(3),10 F(9));
           PGNO = PGNO + 1;
         END;
       ELSE
         PUT FILE(TABLE) PAGE;
     END;

   OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
   SIGNAL ENDPAGE(TABLE);

   PUT FILE(TABLE) EDIT
    ((DEG,(SIND(DEG+MIN) DO MIN = 0 TO .9 BY .1) DO DEG = 0 TO 359))
    (SKIP(2), 5 (COL(1), F(3), 10 F(9,4) ));
   PUT FILE(TABLE) SKIP(52);
 END SINE;
```

Figure 34. Creating a print file via stream data transmission. The example in Figure 38 on page 277 will print the resultant file.

Overriding the tab control table

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions. For 31-bit programs, you can customize the tab control table. However, user-defined PLITAB is not supported for 64-bit programs, which means that you cannot override the tab control table.

See Figure 14 on page 169 and Figure 35 on page 258 for examples of declaring a tab table. The definitions of the fields in the table are as follows:

OFFSET OF TAB COUNT:
: Halfword binary integer that gives the offset of **Tab count**, the field that indicates the number of tabs to be used

PAGESIZE:
: Halfword binary integer that defines the default page size

This page size is used for dump output to the PLIDUMP data set as well as for stream output.

LINESIZE:
Halfword binary integer that defines the default line size

PAGELENGTH:
Halfword binary integer that defines the default page length for printing at a terminal

FILLERS:
Three halfword binary integers, reserved for future use

TAB COUNT:
Halfword binary integer that defines the number of tab position entries in the table (maximum 255)

If tab count = 0, any specified tab positions are ignored.

Tab1–Tabn:
`n` halfword binary integers that define the tab positions within the print line

The first position is numbered 1, and the highest position is numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it is ignored. The first data field in the printed output begins at the next available tab position.

You can override the default PL/I tab settings for your program by causing the linkage editor to resolve an external reference to PLITABS. To cause the reference to be resolved, supply a table with the name PLITABS, in the format described above.

To supply this tab table, include a PL/I structure in your source program with the name PLITABS, which you must declare to be STATIC EXTERNAL in your MAIN procedure or in a program linked with your MAIN procedure. An example of the PL/I structure is shown in Figure 35. This example creates three tab settings, in positions 30, 60, and 90, and uses the defaults for page size and line size. Note that TAB1 identifies the position of the second item printed on a line; the first item on a line always starts at the left margin. The first item in the structure is the offset to the NO_OF_TABS field. The FILL fields must not be omitted.

```
DCL 1 PLITABS STATIC EXT,
    2 (OFFSET INIT(14),
      PAGESIZE INIT(60),
      LINESIZE INIT(120),
      PAGELENGTH INIT(0),
      FILL1 INIT(0),
      FILL2 INIT(0),
      FILL3 INIT(0),
      NO_OF_TABS INIT(3),
      TAB1 INIT(30),
      TAB2 INIT(60),
      TAB3 INIT(90)) FIXED BIN(15,0);
```

Figure 35. PL/I structure PLITABS for modifying the preset tab settings

Using SYSIN and SYSPRINT files for 31-bit programs

If you code a GET statement without the FILE option in your program, the compiler inserts the file name SYSIN. If you code a PUT statement without the FILE option, the compiler inserts the name SYSPRINT.

If you do not declare SYSPRINT, the compiler gives the file the attribute PRINT in addition to the normal default attributes. Here is the complete set of attributes:

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

Because SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 characters and a V-format record. You need give only a minimum of information in the corresponding DD statement; if your installation uses the usual convention that the system output device of class A is a printer, the following is sufficient:

```
//SYSPRINT DD SYSOUT=A
```

Note: SYSIN and SYSPRINT are established in the User Exit during initialization. IBM-supplied defaults for SYSIN and SYSPRINT are directed to the terminal.

You can override the attributes given to SYSPRINT by the compiler by explicitly declaring or opening the file. For more information about the interaction between SYSPRINT and the z/OS Language Environment message file option, see the *z/OS Language Environment Programming Guide*.

The compiler does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes. The data set associated with SYSIN is usually in the input stream; if it is not in the input stream, you must supply full DD information.

Related information:

"SYSPRINT considerations for 31-bit programs" on page 172

Using SYSIN and SYSPRINT files for 64-bit programs

For 64-bit programs, SYSPRINT is equated to the C stdout file; and SYSIN is equated to the C stdin file.

Related information:

"SYSPRINT considerations for 64-bit programs" on page 190
For 64-bit programs, SYSPRINT is equated to the C stdout file. In addition, shared SYSPRINT is not supported.

Controlling input from the terminal

You can enter data at the terminal for an input file in your PL/I program if you do the following:

1. Declare the input file explicitly or implicitly with the CONSECUTIVE environment option (all stream files meet this condition).
2. Allocate the input file to the terminal.

You can usually use the standard default input file SYSIN because it is a stream file and can be allocated to the terminal.

You are prompted for input to stream files by a colon (:). You will see the colon each time a GET statement is executed in the program. The GET statement causes the system to go to the next line. You can then enter the required data. If you enter

a line that does not contain enough data to complete execution of the GET statement, a further prompt, which is a plus sign followed by a colon (+:), is displayed.

By adding a hyphen to the end of any line that is to continue, you can delay transmission of the data to your program until you enter two or more lines.

If you include output statements that prompt you for input in your program, you can inhibit the initial system prompt by ending your own prompt with a colon. For example, the GET statement could be preceded by a PUT statement as follows:

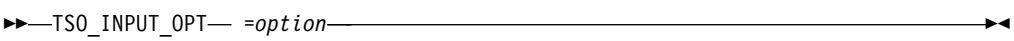
```
PUT SKIP LIST('ENTER NEXT ITEM:');
```

To inhibit the system prompt for the next GET statement, your own prompt must meet the following conditions:

1. It must be either list-directed or edit-directed, and if list-directed, must be to a PRINT file.
2. The file transmitting the prompt must be allocated to the terminal. If you are merely copying the file at the terminal, the system prompt is not inhibited.

Under TSO, there is support of an environment variable called TSO_INPUT_OPT to help control the input from a terminal in a more flexible way than the default way described above.

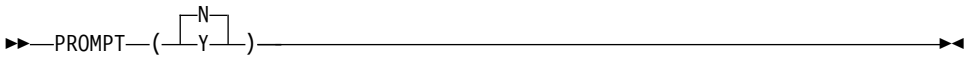
The syntax for the TSO_INPUT_OPT environment variable, which must be in uppercase, is as follows:



When you specify the options, which can be in uppercase or lowercase, blanks are not allowed. If more than one option is specified, they have to be separated by a comma. In addition, the syntax of the statement is not checked at the time the command is entered. It is verified when the data set is opened. If the syntax is wrong, the UNDEFINEDFILE condition is raised with the oncode 96.

You can specify the following options:

PROMPT
The PROMPT option specifies whether a colon should be visible as prompts for stream input from the terminal.



SAMELINE
The SAMELINE option specifies whether the system prompt occurs on the same line as the statement that prompts for input.

For more information about how these options affect your input and output to the terminal, see the examples in "SAMELINE" on page 218.

One way to specify this environment variable under TSO is through the PLIXOPT string. See the following example:

```
DCL PLIXOPT char(50) var ext static
    init('ENVAR("TSO_INPUT_OPT=PROMPT(Y),SAMELINE(Y)")');
```

Format of data

The data you enter at the terminal should have exactly the same format as stream input data in batch mode, but there are a few variations.

- Simplified punctuation for input

  If you enter separate items of input on separate lines, there is no need to enter intervening blanks or commas; the compiler will insert a comma at the end of each line.

  For instance, in response to the statement:

  ```
  GET LIST(I,J,K);
  ```

  your terminal interaction could be as follows:

  ```
  :
  1
  +:2
  +:3
  ```

  with a carriage return following each item. It would be equivalent to:

  ```
  :
  1,2,3
  ```

  If you wish to continue an item onto another line, you must end the first line with a continuation character. Otherwise, for a GET LIST or GET DATA statement, a comma will be inserted, and for a GET EDIT statement, the item will be padded (see next paragraph).
- Automatic padding for GET EDIT

  There is no need to enter blanks at the end of a line of input for a GET EDIT statement. The item you enter will be padded to the correct length.

  For instance, consider the following PL/I statement:

  ```
  GET EDIT(NAME)(A(15));
  ```

  You can enter the five characters, `SMITH`, followed immediately by a carriage return. The item will be padded with 10 blanks, so that the program receives a string 15 characters long. If you want to continue an item on a second or subsequent line, you must add a continuation character to the end of every line except the last; the first line transmitted would otherwise be padded and treated as the complete data item.
- SKIP option or format item

  A SKIP in a GET statement asks the program to ignore data not yet entered. All uses of SKIP(*n*) where *n* is greater than one are taken to mean SKIP(1). SKIP(1) means that all unused data on the current line is ignored.

Stream and record files

You can allocate both stream and record files to the terminal. However, no prompting is provided for record files.

If you allocate more than one file to the terminal and one or more of them is a record file, the output of the files will not necessarily be synchronized. The order

in which data is transmitted to and from the terminal is not guaranteed to be the same order in which the corresponding PL/I I/O statements are executed.

Also, record file input from the terminal is received in uppercase letters because of a TCAM restriction. To avoid problems, you should use stream files wherever possible.

Defining QSAM files using PL/I dynamic allocation

You can define QSAM or HFS files by using a DD statement, an environment variable, or the TITLE option of the OPEN statement.

When an environment variable or the TITLE option is used, the name must be in uppercase. Specify the MVS data set in one of the following ways:

- `DSN(data-set-name)`
- `DSN(data-set-name (member-name))`

data-set-name must be fully qualified and cannot be a temporary data set; for example, it must not start with &.

Specify the HFS file as follows:

```
PATH (absolute-path-name)
```

The following attributes can be specified in any order after the DSN keyword:

```
NEW, OLD, SHR, or MOD
TRACKS or CYL
SPACE(n,m)
VOL(volser)
UNIT(type)
KEEP, DELETE, CATALOG, or UNCATALOG
STORCLAS(storageclass)
MGMTCLAS(managementclass)
DATACLAS(dataclass)
```

Note: You cannot create a PDS or PDSE by using an environment variable or the TITLE option of the OPEN statement, but you can create a new member in an existing PDS or PDSE.

Capital and lowercase letters

For stream files, character strings are transmitted to the program as entered in lowercase or uppercase. For record files, all characters become uppercase.

End-of-file

The characters /* in positions one and two of a line that contains no other characters are treated as an end-of-file marker; that is, they raise the ENDFILE condition.

Under the USS environment, you can also use the key sequence ESC-D as the end-of-file marker. ESC is predefined as a specific character, and you need to use this predefined character in the sequence.

COPY option of GET statement

The GET statement can specify the COPY option; but if the COPY file, as well as the input file, is allocated to the terminal, no copy of the data will be printed.

Chapter 11. Controlling output to the terminal

At your terminal you can obtain data from a PL/I file that meets the following conditions:

1. The file is declared explicitly or implicitly with the CONSECUTIVE environment option. All stream files meet this condition.
2. The file is allocated to the terminal.

The standard print file SYSPRINT generally meets both these conditions.

Format of PRINT files

Data from SYSPRINT or other PRINT files is not normally formatted into pages at the terminal. Three lines are always skipped for PAGE and LINE options and format items. The ENDPAGE condition is normally never raised. SKIP(*n*) where *n* is greater than three causes only three lines to be skipped. SKIP(0) is implemented by backspacing, and should therefore not be used with terminals that do not have a backspace feature.

You can cause a PRINT file to be formatted into pages by inserting a tab control table in your program. The table must be called PLITABS, and its contents are explained in "Overriding the tab control table" on page 257. You must initialize the element PAGELENGTH to the length of page you require—that is, the length of the sheet of paper on which each page is to be printed, expressed as the maximum number of lines that could be printed on it. You must initialize the element PAGESIZE to the actual number of lines to be printed on each page. After the number of lines in PAGESIZE has been printed on a page, ENDPAGE is raised, for which standard system action is to skip the number of lines equal to PAGELENGTH minus PAGESIZE, and then start printing the next page. For other than standard layout, you must initialize the other elements in PLITABS to the values shown in Figure 14 on page 169. You can also use PLITABS to alter the tabulating positions of list-directed and data-directed output. You can use PLITABS for SYSPRINT when you need to format page breaks in ILC applications. Set PAGESIZE to 32767 and use the PUT PAGE statement to control page breaks.

Although some types of terminals have a tabulating facility, tabulating of list-directed and data-directed output is always achieved by transmission of blank characters.

Stream and record files

You can allocate both stream and record files to the terminal. However, if you allocate multiple files to the terminal among which one is SYSPRINT or a record file, the output of the files is not necessarily synchronized.

There is no guarantee that the order in which data is transmitted between the program and the terminal is the same as the order in which the corresponding PL/I output statements are executed.

© Copyright IBM Corp. 1999, 2017

Output from the PUT EDIT command

The format of the output from a PUT EDIT command to a terminal is line mode TPUTs with *Start of field* and *end of field* characters appearing as blanks on the screen.

Chapter 12. Using record-oriented data transmission

PL/I supports various types of data sets with the RECORD attribute. This section covers how to use consecutive data sets.

Table 18 lists the statements and options that you can use to create and access a consecutive data set using record-oriented data transmission.

Table 18. Statements and options allowed for creating and accessing consecutive data sets

File declaration[1]	Valid statements[2] with options you must specify	Other options you can specify
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)

© Copyright IBM Corp. 1999, 2017

Table 18. Statements and options allowed for creating and accessing consecutive data sets (continued)

File declaration[1]	Valid statements[2] with options you must specify	Other options you can specify
SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	

Notes:

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT.
2. The statement `READ FILE (file-reference);` is a valid statement and is equivalent to `READ FILE(file-reference) IGNORE (1);`.

Related information:

"Creating a data set with record I/O" on page 272
When you create a consecutive data set, you must open the associated file for SEQUENTIAL OUTPUT. You can use either the WRITE or LOCATE statement to write records.

Specifying record format

If you give record-format information, it must be compatible with the actual structure of the data set.

For example, if you create a data set with FB-format records, with a record size of 600 bytes and a block size of 3600 bytes, you can access the records as if they were U-format with a maximum block size of 3600 bytes. If you specify a block size of 3500 bytes, your data is truncated.

Defining files using record I/O

You define files for record-oriented data transmission by using a file declaration.

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
             ENVIRONMENT(options);
```

For information about the default file attributes, see Table 15 on page 228. The file attributes are described in the *PL/I Language Reference*. For information about the options of the ENVIRONMENT attribute, see "Specifying ENVIRONMENT options."

Specifying ENVIRONMENT options

This section describes the ENVIRONMENT options that are applicable to consecutive data sets.

The following ENVIRONMENT options are applicable to consecutive data sets:

```
F|FB|FS|FBS|V|VB|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING

CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
CTLASA|CTL360
LEAVE|REREAD
```

See the following topics for information about these options.

ENVIRONMENT options	See these topics
F | FB | FS | FBS | V | VB | U	• "The ENVIRONMENT attribute" on page 228 • "Record formats for record-oriented data transmission" on page 230
RECSIZE(record-length)	• "The ENVIRONMENT attribute" on page 228 • "RECSIZE option" on page 231
BLKSIZE(block-size)	• "The ENVIRONMENT attribute" on page 228 • "BLKSIZE option" on page 232
SCALARVARYING	• "The ENVIRONMENT attribute" on page 228 • "SCALARVARYING option — varying-length strings" on page 235
CONSECUTIVE	"CONSECUTIVE"
ORGANIZATION(CONSECUTIVE)	"ORGANIZATION(CONSECUTIVE)"
CTLASA | CTL360	"CTLASA | CTL360" on page 270
LEAVE | REREAD	"LEAVE | REREAD" on page 271

See Table 15 on page 228 to find which options you must specify, which are optional, and which are defaults.

CONSECUTIVE

The CONSECUTIVE option defines a file with consecutive data set organization.

CONSECUTIVE is the default.

Related information:

"Data set organization" on page 223
The data management routines of the operating system can handle a number of types of data sets, which differ in the way data is stored within them and in the allowed means of access to the data.

ORGANIZATION(CONSECUTIVE)

The ORGANIZATION(CONSECUTIVE) option specifies that the file is associated with a consecutive data set.

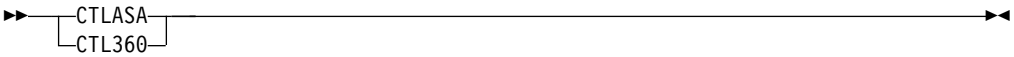
The file can be either a native data set or a VSAM data set.

Related information:

"ORGANIZATION option" on page 236
The ORGANIZATION option specifies the organization of the data set associated with the PL/I file.

CTLASA|CTL360

The printer control options CTLASA and CTL360 apply only to OUTPUT files associated with consecutive data sets. They specify that the first character of a record is to be interpreted as a control character.



The CTLASA option specifies American National Standard Vertical Carriage Positioning Characters or American National Standard Pocket Select Characters (Level 1). The CTL360 option specifies IBM machine-code control characters.

The American National Standard control characters, listed in Figure 36, cause the specified action to occur before the associated record is printed or punched.

The machine code control characters differ according to the type of device. The IBM machine code control characters for printers are listed in Table 19 on page 271.

Code	Action
	Space 1 line before printing (blank code)
0	Space 2 lines before printing
-	Space 3 lines before printing
+	Suppress space before printing
1	Skip to channel 1
2	Skip to channel 2
3	Skip to channel 3
4	Skip to channel 4
5	Skip to channel 5
6	Skip to channel 6
7	Skip to channel 7
8	Skip to channel 8
9	Skip to channel 9
A	Skip to channel 10
B	Skip to channel 11
C	Skip to channel 12
V	Select stacker 1
W	Select stacker 2

Figure 36. American National Standard print and card punch control characters (CTLASA)

Table 19. IBM machine code print control characters (CTL360)

Print and Then Act	Action	Act immediately (no printing)
Code byte		**Code byte**
00000001	Print only (no space)	—
00001001	Space 1 line	00001011
00010001	Space 2 lines	00010011
00011001	Space 3 lines	00011011
10001001	Skip to channel 1	10001011
10010001	Skip to channel 2	10010011
10011001	Skip to channel 3	10011011
10100001	Skip to channel 4	10100011
10101001	Skip to channel 5	10101011
10110001	Skip to channel 6	10110011
10111001	Skip to channel 7	10111011
11000001	Skip to channel 8	11000011
11001001	Skip to channel 9	11001011
11010001	Skip to channel 10	11010011
11011001	Skip to channel 11	11011011
11100001	Skip to channel 12	11100011

LEAVE|REREAD

The magnetic tape handling options LEAVE and REREAD specify the action to be taken when the end of a magnetic tape volume is reached, or when a data set on a magnetic tape volume is closed.

The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to allow reprocessing of the data set. If you do not specify either of these, the action at end-of-volume or on closing of a data set is controlled by the DISP parameter of the associated DD statement.

If a data set is first read or written forward and then read backward in the same program, specify the LEAVE option to prevent rewinding when the file is closed (or, with a multivolume data set, when volume switching occurs).

Table 20 summarizes the effects of the LEAVE and REREAD options.

Table 20. Effect of LEAVE and REREAD Options

ENVIRONMENT option	DISP parameter	Action
REREAD	—	Positions the current volume to reprocess the data set.
LEAVE	—	Positions the current volume at the logical end of the data set.

Table 20. Effect of LEAVE and REREAD Options (continued)

ENVIRONMENT option	DISP parameter	Action
Neither REREAD nor LEAVE	PASS	Positions the volume at the end of the data set.
	DELETE	Rewinds the current volume.
	KEEP CATLG UNCATLG	Rewinds and unloads the current volume.

Creating a data set with record I/O

When you create a consecutive data set, you must open the associated file for SEQUENTIAL OUTPUT. You can use either the WRITE or LOCATE statement to write records.

Table 18 on page 267 shows the statements and options for creating a consecutive data set.

When creating a data set, you must identify it to the operating system in a DD statement. Table 21 summarizes the essential information that you must include in the DD statement and the optional information that you can supply.

Table 21. Creating a consecutive data set with record I/O: essential parameters of the DD statement

Storage device	When required	What you must state	Parameters
All	Always	Output device	UNIT= or SYSOUT= or VOLUME=REF=
		Block size[1]	DCB=(BLKSIZE=...
Direct access only	Always	Storage space required	SPACE=
Direct access	Data set to be used by another job step but not required at end of job	Disposition	DISP=
	Data set to be kept after end of job	Disposition	DISP=
		Name of data set	DSNAME=
	Data set to be on particular device	Volume serial number	VOLUME=SER= or VOLUME=REF=

Note:

1. Or you can specify the block size in your PL/I program by using the ENVIRONMENT attribute.

Essential information

When creating a data set, you must identify it to the operating system in a DD statement. This topic describes the essential information that you must include in the statement.

When you create a consecutive data set, you must specify the following essential information in the DD statement:

- The name of data set to be associated with your PL/I file

A data set with consecutive organization can exist on any type of device.

- The record length

  You can specify the record length by using the RECSIZE option of the ENVIRONMENT attribute, of the DD_DDNAME environment variable, or of the TITLE option of the OPEN statement.

  For files associated with the terminal device (stdout: or stderr:), PL/I uses a default record length of 120 when the RECSIZE option is not specified.

Accessing and updating a data set with record I/O

After you create a consecutive data set, you can open the file that accesses it for sequential input, for sequential output, or, for data sets on direct access devices, for updating.

See Figure 37 on page 275 for an example of a program that accesses and updates a consecutive data set. If you open the file for output and extend the data set by adding records at the end, you must specify DISP=MOD in the DD statement. If you do not, the data set will be overwritten. If you open a file for updating, you can update only records in their existing sequence, and if you want to insert records, you must create a new data set. Table 18 on page 267 shows the statements and options for accessing and updating a consecutive data set.

When you access a consecutive data set by a SEQUENTIAL UPDATE file, you must retrieve a record with a READ statement before you can update it with a REWRITE statement; however, every record that is retrieved need not be rewritten. A REWRITE statement will always update the last record read.

Consider the following statement:

```
READ FILE(F) INTO(A);
     .
     .
     .
READ FILE(F) INTO(B);
     .
     .
     .
REWRITE FILE(F) FROM(A);
```

The REWRITE statement updates the record that was read by the second READ statement. The record that was read by the first statement cannot be rewritten after the second READ statement has been executed.

You cannot update a consecutive data set on magnetic tape except by adding records at the end. To replace or insert records, you must read the data set and write the updated records into a new data set.

You can read a consecutive data set on magnetic tape forward only. Reading backward is not supported.

To access a data set, you must identify it to the operating system in a DD statement. Table 22 on page 274 summarizes the DD statement parameters needed to access a consecutive data set.

Table 22. Accessing a consecutive data set with record I/O: essential parameters of the DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	Always
UNIT= or VOLUME=REF=	Input device	If data set not cataloged (all devices)
VOLUME=SER=	Volume serial number	If data set not cataloged (direct access)
DCB=(BLKSIZE=	Block size[1]	If data set does not have standard labels

Note:

1. Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

The following topics describe the essential information that you must include in the DD statement, and discuss some of the optional information that you can supply. The discussions do not apply to data sets in the input stream.

Essential information

If the data set is cataloged, you need to supply only the following information in the DD statement:

- The name of the data set (DSNAME parameter)

  The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.
- Confirmation that the data set exists (DISP parameter)

  If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, opening the data set for output will result in it being overwritten.

If the data set is not cataloged, you must additionally specify the device that will read the data set, and for direct access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

Example of consecutive data sets

Example: Merge Sort—creating and accessing a consecutive data set

Figure 37 on page 275 illustrates creating and accessing consecutive data sets. The program merges the contents of two data sets, in the input stream, and writes them onto a new data set, &&TEMP; each of the original data sets contains 15-byte fixed-length records arranged in EBCDIC collating sequence. The two input files, INPUT1 and INPUT2, have the default attribute BUFFERED, and locate mode is used to read records from the associated data sets into the respective buffers. Access of based variables in the buffers should not be attempted after the file has been closed.

```
//EXAMPLE  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN   DD *
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 MERGE: PROC OPTIONS(MAIN);
   DCL (INPUT1,                              /* FIRST INPUT FILE     */
        INPUT2,                              /* SECOND INPUT FILE    */
        OUT )      FILE RECORD SEQUENTIAL;   /* RESULTING MERGED FILE*/
   DCL SYSPRINT    FILE PRINT;               /* NORMAL PRINT FILE    */

   DCL INPUT1_EOF  BIT(1) INIT('0'B);        /* EOF FLAG FOR INPUT1  */
   DCL INPUT2_EOF  BIT(1) INIT('0'B);        /* EOF FLAG FOR INPUT2  */
   DCL OUT_EOF     BIT(1) INIT('0'B);        /* EOF FLAG FOR OUT     */
   DCL TRUE        BIT(1) INIT('1'B);        /* CONSTANT TRUE        */
   DCL FALSE       BIT(1) INIT('0'B);        /* CONSTANT FALSE       */

   DCL ITEM1       CHAR(15) BASED(A);        /* ITEM FROM INPUT1     */
   DCL ITEM2       CHAR(15) BASED(B);        /* ITEM FROM INPUT2     */
   DCL INPUT_LINE  CHAR(15);                 /* INPUT FOR READ INTO  */
   DCL A           POINTER;                  /* POINTER VAR          */
   DCL B           POINTER;                  /* POINTER VAR          */

   ON ENDFILE(INPUT1) INPUT1_EOF = TRUE;
   ON ENDFILE(INPUT2) INPUT2_EOF = TRUE;
   ON ENDFILE(OUT)    OUT_EOF    = TRUE;

   OPEN FILE(INPUT1) INPUT,
        FILE(INPUT2) INPUT,
        FILE(OUT)    OUTPUT;

   READ FILE(INPUT1) SET(A);                 /* PRIMING READ         */
   READ FILE(INPUT2) SET(B);

   DO WHILE ((INPUT1_EOF = FALSE) & (INPUT2_EOF = FALSE));
     IF ITEM1 > ITEM2 THEN
       DO;
         WRITE FILE(OUT) FROM(ITEM2);
         PUT FILE(SYSPRINT) SKIP EDIT('1>2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT2) SET(B);
       END;
     ELSE
       DO;
         WRITE FILE(OUT) FROM(ITEM1);
         PUT FILE(SYSPRINT) SKIP EDIT('1<2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT1) SET(A);
       END;
   END;
```

Figure 37. Merge Sort—creating and accessing a consecutive data set

```
   DO WHILE (INPUT1_EOF = FALSE);           /* INPUT2 IS EXHAUSTED  */
     WRITE FILE(OUT) FROM(ITEM1);
     PUT FILE(SYSPRINT) SKIP EDIT('1', ITEM1) (A(2),A);
     READ FILE(INPUT1) SET(A);
   END;

   DO WHILE (INPUT2_EOF = FALSE);           /* INPUT1 IS EXHAUSTED  */
     WRITE FILE(OUT) FROM(ITEM2);
     PUT FILE(SYSPRINT) SKIP EDIT('2', ITEM2) (A(2),A);
     READ FILE(INPUT2) SET(B);
   END;

   CLOSE FILE(INPUT1), FILE(INPUT2), FILE(OUT);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(OUT) SEQUENTIAL INPUT;

   READ FILE(OUT) INTO(INPUT_LINE);         /* DISPLAY OUT FILE     */
   DO WHILE (OUT_EOF = FALSE);
     PUT FILE(SYSPRINT) SKIP EDIT(INPUT_LINE) (A);
     READ FILE(OUT) INTO(INPUT_LINE);
   END;
   CLOSE FILE(OUT);

 END MERGE;
/*
//GO.INPUT1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.INPUT2 DD *
BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,
//          DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))
```

Merge Sort—creating and accessing a consecutive data set (continued)

Example: Printing record-oriented data transmission

The program in Figure 38 on page 277 uses record-oriented data transmission to print the table created by the program in Figure 34 on page 257.

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 PRT: PROC OPTIONS(MAIN);
   DCL TABLE       FILE RECORD INPUT SEQUENTIAL;
   DCL PRINTER     FILE RECORD OUTPUT SEQL
                        ENV(V BLKSIZE(102) CTLASA);
   DCL LINE        CHAR(94) VAR;

   DCL TABLE_EOF   BIT(1) INIT('0'B);       /* EOF FLAG FOR TABLE   */
   DCL TRUE        BIT(1) INIT('1'B);       /* CONSTANT TRUE        */
   DCL FALSE       BIT(1) INIT('0'B);       /* CONSTANT FALSE       */


   ON ENDFILE(TABLE) TABLE_EOF = TRUE;

   OPEN FILE(TABLE),
        FILE(PRINTER);

   READ FILE(TABLE) INTO(LINE);             /* PRIMING READ         */

   DO WHILE (TABLE_EOF = FALSE);
     WRITE FILE(PRINTER) FROM(LINE);
     READ FILE(TABLE) INTO(LINE);
   END;

   CLOSE FILE(TABLE),
         FILE(PRINTER);
 END PRT;
```

Figure 38. Printing record-oriented data transmission

Chapter 13. Defining and using regional data sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. It also includes information about how to create and access regional data sets for each type of regional organization.

Note: Regional data sets are not supported for 64-bit programs in PL/I V5.1.

A data set with regional organization is divided into regions, each of which is identified by a region number, and each of which can contain one record or more than one record, depending on the type of regional organization. The regions are numbered in succession, beginning with zero, and a record can be accessed by specifying its region number, and perhaps a key, in a data transmission statement.

Regional data sets are confined to direct access devices.

Regional organization of a data set allows you to control the physical placement of records in the data set, and to optimize the access time for a particular application. Such optimization is not available with consecutive or indexed organization, in which successive records are written either in strict physical sequence or in logical sequence depending on ascending key values; neither of these methods takes full advantage of the characteristics of direct access storage devices.

You can create a regional data set in a manner similar to a consecutive or indexed data set, presenting records in the order of ascending region numbers; alternatively, you can use direct access, in which you present records in random sequence and insert them directly into preformatted regions. After you create a regional data set, you can access it by using a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. You do not need to specify either a region number or a key if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Records within a regional data set are either actual records containing valid data or dummy records.

The major advantage of regional organization over other types of data set organization is that it allows you to control the relative placement of records; by judicious programming, you can optimize record access in terms of device capabilities and the requirements of particular applications.

Direct access of regional data sets is quicker than that of indexed data sets, but regional data sets have the disadvantage that sequential processing can present records in random sequence; the order of sequential retrieval is not necessarily that in which the records were presented, nor need it be related to the relative key values.

Table 23 on page 280 lists the data transmission statements and options that you can use to create and access a regional data set.

© Copyright IBM Corp. 1999, 2017

Table 23. Statements and options allowed for creating and accessing regional data sets

File declaration[1]	Valid statements[2] with options you must include	Other options you can also include
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE based-variable FROM(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE[3]	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	
DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	DELETE FILE(file-reference) KEY(expression);	

Notes:

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED.
2. The statement `READ FILE(file-reference);` is equivalent to the statement `READ FILE(file-reference) IGNORE(1);`.
3. When you create new data sets, the file must not have the UPDATE attribute.

Defining REGIONAL(1) data sets using PL/I dynamic allocation

You can define REGIONAL(1) data sets by using a DD statement, an environment variable, or the TITLE option of the OPEN statement.

When an environment variable or the TITLE option is used, the name must be in uppercase. Specify the MVS data set as follows:

```
DSN(data-set-name)
```

data-set-name must be fully qualified and cannot be a temporary data set; for example, it must not start with &.

You must specify one of the following attributes after the DSN keyword:

```
OLD
SHR
```

Defining files for a regional data set

You can use a file declaration to define a regional data set.

Defining a sequential regional data set

To define a sequential regional data set, use a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

Because BUFFERED and UNBUFFERED will be treated the same for REGIONAL(1) data sets, you can specify either option in the ENVIRONMENT option. For example, the FROM option is not required on a REWRITE for a SEQUENTIAL UNBUFFERED file and the LOCATE statement is allowed for OUTPUT SEQUENTIAL data sets even if UNBUFFERED is specified.

Defining a direct regional data set

To define a direct regional data set, use a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             ENVIRONMENT(options);
```

For information about the default file attributes, see Table 15 on page 228. For detailed information about the file attributes, see the *PL/I Language Reference*. For information about the suboptions of the ENVIRONMENT option, see "Specifying ENVIRONMENT options."

Specifying ENVIRONMENT options

This section describes ENVIRONMENT options that are applicable to regional data sets.

The following ENVIRONMENT options are applicable to regional data sets:

```
REGIONAL({1})
F
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
```

REGIONAL

You can use the REGIONAL option to define a file with regional organization.

1 Specifies REGIONAL(1).

REGIONAL(1)
Specifies that the data set contains F-format records that do not have recorded keys. Each region in the data set contains only one record; therefore, each region number corresponds to a relative record within the data set (that is, region numbers start with 0 at the beginning of the data set).

Although REGIONAL(1) data sets have no recorded keys, you can use REGIONAL(1) DIRECT INPUT or UPDATE files to process data sets that do have recorded keys.

RECSIZE(record-length)
BLKSIZE(block-size)
If both RECSIZE and BLKSIZE are specified, they must specify the same value.

REGIONAL(1) organization is most suited to applications where there are no duplicate region numbers, and where most of the regions will be filled (reducing wasted space in the data set).

Using keys with REGIONAL data sets

There are two kinds of keys, *recorded keys* and *source keys*.

A *recorded key* is a character string that immediately precedes each record in the data set to identify that record; its length cannot exceed 255 characters. A *source key* is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. When you access a record in a regional data set, the source key gives a region number, and can also give a recorded key.

Unlike the keys for indexed data sets, recorded keys in a regional data set are never embedded within the record.

Using REGIONAL(1) data sets

In a REGIONAL(1) data set, because there are no recorded keys, the region number serves as the sole identification of a particular record.

The character value of the source key should represent an unsigned decimal integer that should not exceed 16777215, although the actual number of records allowed can be smaller, depending on a combination of record size, device capacity, and limits of your access method. For direct regional(1) files with fixed format records, the maximum number of tracks that can be addressed by relative track addressing is 65536. If the region number exceeds this figure, it is treated as modulo 16777216; for instance, 16777226 is treated as 10. Only the characters 0 through 9 and the blank character are valid in the source key; leading blanks are

interpreted as zeros. Embedded blanks are not allowed in the number; the first embedded blank, if any, terminates the region number. If more than 8 characters appear in the source key, only the rightmost 8 are used as the region number; if there are fewer than 8 characters, blanks (interpreted as zeros) are inserted on the left.

Dummy Records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant (8)'1'B in its first byte.

Although such dummy records are inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read; your PL/I program must be prepared to recognize them. You can replace dummy records with valid data. Note that if you insert (8)'1'B in the first byte, the record can be lost if you copy the file onto a data set that has dummy records that are not retrieved.

Creating a REGIONAL(1) data set

You can create a REGIONAL(1) data set either sequentially or by direct access.

Table 23 on page 280 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, the opening of the file causes all tracks on the data set to be cleared, and a capacity record to be written at the beginning of each track to record the amount of space available on that track. You must present records in ascending order of region numbers; any region you omit from the sequence is filled with a dummy record. If there is an error in the sequence, or if you present a duplicate key, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If you use a DIRECT OUTPUT file to create the data set, the whole primary extent allocated to the data set is filled with dummy records when the file is opened. You can present records in random order; if you present a duplicate, the existing record will be overwritten.

For sequential creation, the data set can have up to 15 extents, which can be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

Example

Figure 39 on page 284 illustrates how to create a REGIONAL(1) data set. The data set is a list of telephone numbers with the names of the subscribers to whom they are allocated. The telephone numbers correspond with the region numbers in the data set, the data in each occupied region being a subscriber's name.

```
//EX9    JOB
//STEP1  EXEC IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 CRR1:   PROC OPTIONS(MAIN);
  /*  CREATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY     */

   DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
   DCL  SYSIN FILE INPUT RECORD;
   DCL  SYSIN_REC BIT(1) INIT('1'B);
   DCL 1   CARD,
        2   NAME   CHAR(20),
        2   NUMBER CHAR( 2),
        2   CARD_1 CHAR(58);
   DCL IOFIELD CHAR(20);

      ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
      OPEN FILE(NOS);
      READ FILE(SYSIN) INTO(CARD);

      DO WHILE(SYSIN_REC);
         IOFIELD = NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         READ FILE(SYSIN) INTO(CARD);
      END;

      CLOSE FILE(NOS);
  END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS     DD DSN=MYID.NOS,UNIT=SYSDA,SPACE=(20,100),
//           DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G.           12
BAKER,R.            13
BRAMLEY,O.H.        28
CHEESNAME,L.        11
CORY,G.             36
ELLIOTT,D.          85
FIGGINS,E.S.        43
HARVEY,C.D.W.       25
HASTINGS,G.M.       31
KENDALL,J.G.        24
LANCASTER,W.R.      64
MILES,R.            23
NEWMAN,M.W.         40
PITT,W.H.           55
ROLF,D.E.           14
SHEERS,C.D.         21
SURCLIFFE,M.        42
TAYLOR,G.C.         47
WILTON,L.W.         44
WINSTONE,E.M.       37
/*
```

Figure 39. Creating a REGIONAL(1) data set

Accessing and updating a REGIONAL(1) data set

After you create a REGIONAL(1) data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. You can open it for OUTPUT only if the existing data set is to be overwritten.

Table 23 on page 280 shows the statements and options for accessing a regional data set.

Sequential access

To open a SEQUENTIAL file that is used to process a REGIONAL(1) data set, use either the INPUT or UPDATE attribute.

You must not include the KEY option in data transmission statements, but the file can have the KEYED attribute, because you can use the KEYTO option. If the target character string referenced in the KEYTO option has more than 8 characters, the value returned (the 8-character region number) is padded on the left with blanks. If the target string has fewer than 8 characters, the value returned is truncated on the left.

Sequential access is in the order of ascending region numbers. All records are retrieved, whether dummy or actual, and you must ensure that your PL/I program recognizes dummy records.

Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region-number sequence, and in sequential update you can read and rewrite each record in turn.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(1) data set are identical to those for a consecutive data set.

Related information:

Chapter 10, “Defining and using consecutive data sets,” on page 245
This chapter covers consecutive data set organization and the ENVIRONMENT options that define consecutive data sets for stream and record-oriented data transmission. It then covers how to create, access, and update consecutive data sets for each type of transmission.

Direct access

To open a DIRECT file that is used to process a REGIONAL(1) data set, you can use either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

Use DIRECT UPDATE files to retrieve, add, delete, or replace records in a REGIONAL(1) data set according to the following conventions:

Retrieval
: All records, whether dummy or actual, are retrieved. Your program must recognize dummy records.

Addition
: A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key.

Deletion
: The record you specify by the source key in a DELETE statement is converted to a dummy record.

Replacement
: The record you specify by the source key in a REWRITE statement, whether dummy or actual, is replaced.

Example

This example illustrates how to update a REGIONAL(1) data set.

The program shown in Figure 40 updates the data set and lists its contents. Before each new or updated record is written, the existing record in the region is tested to ensure that it is a dummy; this is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

```
//EX10    JOB
//STEP2   EXEC  IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 ACR1: PROC OPTIONS(MAIN);
   /*  UPDATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY      */
    DCL NOS FILE RECORD  KEYED ENV(REGIONAL(1));
    DCL  SYSIN FILE INPUT RECORD;
    DCL  (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
    DCL 1   CARD,
         2   NAME   CHAR(20),
         2   (NEWNO,OLDNO)  CHAR( 2),
         2   CARD_1 CHAR( 1),
         2   CODE   CHAR( 1),
         2   CARD_2 CHAR(54);
    DCL IOFIELD CHAR(20);
    DCL BYTE    CHAR(1) DEF IOFIELD;

   ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
   OPEN FILE (NOS) DIRECT UPDATE;
   READ FILE(SYSIN) INTO(CARD);

   DO WHILE(SYSIN_REC);
      SELECT(CODE);
         WHEN('A','C') DO;
             IF CODE = 'C' THEN
                DELETE FILE(NOS) KEY(OLDNO);
             READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
             IF UNSPEC(BYTE) = (8)'1'B
                THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
             ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
         END;
         WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
      OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
      END;
      READ FILE(SYSIN) INTO(CARD);
   END;

   CLOSE FILE(SYSIN),FILE(NOS);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(NOS) SEQUENTIAL INPUT;
   ON ENDFILE(NOS) NOS_REC = '0'B;
   READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   DO WHILE(NOS_REC);
      IF UNSPEC(BYTE) ¬= (8)'1'B
         THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD)(A(2),X(3),A);
      PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   END;
  CLOSE FILE(NOS);
  END ACR1;
/*
```

Figure 40. Updating a REGIONAL(1) data set

```
//GO.NOS   DD DSN=J44PLI.NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.         5640 C
GOODFELLOW,D.T.     89   A
MILES,R.              23 D
HARVEY,C.D.W.       29   A
BARTLETT,S.G.       13   A
CORY,G.               36 D
READ,K.M.           01   A
PITT,W.H.             55
ROLF,D.F.             14 D
ELLIOTT,D.          4285 C
HASTINGS,G.M.         31 D
BRAMLEY,O.H.        4928 C
/*
```

Updating a REGIONAL(1) data set (continued)

Essential information for creating and accessing regional data sets

To create a regional data set, in your PL/I program or in the DD statement, you must give the operating system certain information that defines the data set.

You must supply the following information when creating a regional data set:

- Device that will write your data set (UNIT or VOLUME parameter of DD statement)
- Block size

  You can specify the block size either in your PL/I program (in the BLKSIZE option of the ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size. If you do specify a record length, it must be equal to the block size.

If you want to keep a data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but do not need it after the end of your job.

If you want your data set stored on a particular direct access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system allocates one, informs the operator, and prints the number on your program listing.

Table 24 on page 288 summarizes all the essential parameters required in a DD statement to create a regional data set. Table 25 on page 289 lists the DCB subparameters that are needed. See your *z/OS MVS JCL User's Guide* for a description of the DCB subparameters.

You cannot place a regional data set on a system output (SYSOUT) device.

In the DCB parameter, if you specify the DSORG parameter, you must specify the data set organization as direct by coding DSORG=DA. You cannot specify the

DUMMY or DSN=NULLFILE parameters in a DD statement for a regional data set. Using DSORG=DA might cause message IEC225I to be issued. This message can be safely ignored.

Table 24. Creating a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
UNIT= or VOLUME=REF=	Output device[1]	Always
SPACE=	Storage space required[2]	Always
DCB=	Data control block information See Table 25 on page 289.	Always
DISP=	Disposition	Data set to be used in another job step but not required in another job
DISP=	Disposition	Data set to be kept after end of job
DSNAME=	Name of data set	
VOLUME=SER= or VOLUME=REF=	Volume serial number	Data set to be on particular volume

Notes:

1. Regional data sets are confined to direct access devices.
2. For sequential access, the data set can have up to 15 extents, which can be on more than one volume. For creation with DIRECT access, the data set can have only one extent.

To access a regional data set, you must identify it to the operating system in a DD statement. The following paragraphs indicate the minimum information you must include in the DD statement; this information is summarized in Table 26 on page 289.

If the data set is cataloged, you need to supply only the following information in your DD statement:

- The name of the data set (DSNAME parameter)

  The operating system locates the information that describes the data set in the system catalog and, if necessary, requests the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter)

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

When opening a multiple-volume regional data set for sequential update, the ENDFILE condition is raised at the end of the first volume.

Table 25. DCB subparameters for a regional data set

Subparameters	To specify	When required
RECFM=F	Record format[1]	Always
BLKSIZE=	Block size[1]	
DSORG=DA	Data set organization	

[1]Or you can specify the block size in the ENVIRONMENT attribute.

Table 26. Accessing a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	
UNIT= or VOLUME=REF=	Input device	If data set not cataloged
VOLUME=SER=	Volume serial number	

Chapter 14. Defining and using VSAM data sets

This chapter covers VSAM (the Virtual Storage Access Method) organization for record-oriented data transmission, VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and the statements you use to load and access the three types of VSAM data sets that PL/I supports—entry-sequenced, key-sequenced, and relative record.

The chapter is concluded by a series of examples showing the PL/I statements, Access Method Services commands, and JCL statements necessary to create and access VSAM data sets.

Enterprise PL/I provides no support for ISAM datasets.

For additional information about the facilities of VSAM, the structure of VSAM data sets and indexes, the way in which they are defined by Access Method Services, and the required JCL statements, see the VSAM publications for your system.

Defining VSAM file using PL/I dynamic allocation

You can define VSAM data sets by using a DD statement, an environment variable, or the TITLE option of the OPEN statement.

When an environment variable or the TITLE option is used, the name must be in uppercase. Specify the MVS data set as follows:

```
DSN(data-set-name)
```

data-set-name must be fully qualified and cannot be a temporary data set; for example, it must not start with &.

You must specify one of the following attributes after the DSN keyword:

```
OLD
SHR
```

Using VSAM data sets

If your program needs to use VSAM data sets, you must specify a DD statement to give the program access to the data sets. Your program can also access key sequenced and entry sequenced data sets through alternate index paths.

Running a program with VSAM data sets

To allow your program to access VSAM data sets, you must provide certain information in your DD statement.

Before you execute a program that accesses a VSAM data set, you need to know the following information:

- The name of the VSAM data set
- The name of the PL/I file
- Whether you intend to share the data set with other users

Then you can write the required DD statement to access the data set:

© Copyright IBM Corp. 1999, 2017

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

For example, if your file is named PL1FILE, your data set is named VSAMDS, and you want exclusive control of the data set, enter the following statement:

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

To share your data set, use DISP=SHR.

Enterprise PL/I has no support for ISAM data sets.

To optimize VSAM's performance by controlling the number of VSAM buffers used for your data set, see the VSAM publications.

Pairing an alternate index path with a file

When using an alternate index, you simply specify the name of the *path* in the DSNAME parameter of the DD statement associating the base data set/alternate index pair with your PL/I file.

Before using an alternate index, you must be aware of the restrictions on processing; these are summarized in Table 30 on page 297.

Given a PL/I file called PL1FILE and the alternate index path called PERSALPH, the DD statement required is as follows:

```
//PL1FILE DD DSNAME=PERSALPH,DISP=OLD
```

VSAM organization

There are three types of VSAM data sets. Each type roughly corresponds to a PL/I data set organization. All three types of VSAM data sets are ordered, and they can all have keys associated with their records. Both sequential and keyed access are possible with all three types.

Table 27. Types of VSAM data sets and corresponding PL/I data set organization

VSAM data set type	Corresponding PL/I data set organization
Key-sequenced data sets (KSDS)	Indexed data set
Entry-sequenced data sets (ESDS)	Consecutive data set
Relative record data sets (RRDS)	Regional data set

Although only key-sequenced data sets have keys as part of their logical records, keyed access is also possible for entry-sequenced data sets (using relative-byte addresses) and relative record data sets (using relative record numbers).

All VSAM data sets are held on direct access storage devices, and a virtual storage operating system is required to use them.

The physical organization of VSAM data sets differs from those used by other access methods. VSAM does not use the concept of blocking, and, except for relative record data sets, records need not be of a fixed length. In data sets with VSAM organization, the data items are arranged in *control intervals*, which are in turn arranged in *control areas*. For processing purposes, the data items within a control interval are arranged in logical records. A control interval can contain one or more logical records, and a logical record can span two or more control intervals. Concern about blocking factors and record length is largely removed by

VSAM, although records cannot exceed the maximum specified size. VSAM allows access to the control intervals, but this type of access is not supported by PL/I.

VSAM data sets can have two types of indexes—prime and alternate. A *prime index* is the index to a KSDS that is established when you define a data set; it always exists and can be the only index for a KSDS. You can have one or more *alternate indexes* on a KSDS or an ESDS. Defining an *alternate index* for an ESDS enables you to treat the ESDS, in general, as a KSDS. An alternate index on a KSDS enables a field in the logical record different from that in the prime index to be used as the key field. Alternate indexes can be either *nonunique*, in which duplicate keys are allowed, or *unique*, in which they are not. The prime index can never have duplicate keys.

Any change in a data set that has alternate indexes must be reflected in all the indexes if they are to remain useful. This activity is known as *index upgrade*, and is done by VSAM for any index in the *index upgrade set* of the data set. (For a KSDS, the prime index is always a member of the index upgrade set.) However, you must avoid making changes in the data set that would cause duplicate keys in the prime index or in a unique alternate index.

Before using a VSAM data set for the first time, you need to define it to the system with the DEFINE command of Access Method Services, which you can use to completely define the type, structure, and required space of the data set. This command also defines the data set's indexes (together with their key lengths and locations) and the index upgrade set if the data set is a KSDS or has one or more alternate indexes. A VSAM data set is thus “created” by Access Method Services.

The operation of writing the initial data into a newly created VSAM data set is referred to as *loading* in this publication.

Use the three different types of data sets according to the following purposes:

- Use *entry-sequenced data sets* for data that you primarily access in the order in which it was created (or the reverse order).
- Use *key-sequenced data sets* when you normally access records through keys within the records (for example, a stock-control file where the part number is used to access a record).
- Use *relative record data sets* for data in which each item has a particular number, and you normally access the relevant record by that number (for example, a telephone system with a record associated with each number).

You can access records in all types of VSAM data sets either directly by a key, or sequentially (backward or forward). You can also use a combination of the two ways: Select a starting point with a key and then read forward or backward from that point.

You can create *alternate indexes* for key-sequenced and entry-sequenced data sets. You can then access your data in many sequences or by one of many keys. For example, you could take a data set held or indexed in order of employee number and index it by name in an *alternate index*. Then you could access it in alphabetic order, in reverse alphabetic order, or directly by using the name as a key. You could also access it in the same kind of combinations by employee number.

Table 28 on page 294 shows how the same data could be held in the three different types of VSAM data sets and illustrates their respective advantages and disadvantages.

Table 28. Types and advantages of VSAM data sets

Data set type	Method of loading	Method of reading	Method of updating	Pros and cons
Key-Sequenced	Sequentially in order or prime index which must be unique	KEYED by specifying key of record in prime index SEQUENTIAL backward or forward in order of any index Positioning by key followed by sequential reading either backward or forward	KEYED specifying a unique key in any index SEQUENTIAL following positioning by unique key Record deletion allowed Record insertion allowed	***Advantages***: Complete access and updating ***Disadvantages***: Records must be in order of prime index before loading ***Uses***: For uses where access will be related to key
Entry-Sequenced	Sequentially (forward only) The RBA of each record can be obtained and used as a key	SEQUENTIAL backward or forward KEYED using RBA Positioning by key followed by sequential either backward or forward	New records at end only Existing records cannot have length changed Record deletion not allowed	***Advantages***: Simple fast creation No requirement for a unique index ***Disadvantages***: Limited updating facilities ***Uses***: For uses where data will primarily be accessed sequentially
Relative Record	Sequentially starting from slot 1 KEYED specifying number of slot Positioning by key followed by sequential writes	KEYED specifying numbers as key Sequential forward or backward omitting empty records	Sequentially starting at a specified slot and continuing with next slot Keyed specifying numbers as key Record deletion allowed Record insertion into empty slots allowed	***Advantages***: Speedy access to record by number ***Disadvantages***: Structure tied to numbering sequences Fixed length records ***Uses***: For use where records will be accessed by number

Keys for VSAM data sets

All VSAM data sets can have keys associated with their records.

For key-sequenced data sets, and for entry-sequenced data sets accessed through an *alternate index*, the key is a defined field within the logical record. For entry-sequenced data sets, the key is the *relative byte address* (RBA) of the record. For relative-record data sets, the key is a *relative record number*.

Keys for indexed VSAM data sets

Keys for key-sequenced data sets and for entry-sequenced data sets that are accessed through an *alternate index* are part of the logical records recorded on the data set. You define the length and location of the keys when you create the data set.

For information about how you can reference the keys in the KEY, KEYFROM, and KEYTO options, see the topics about the KEY(expression) option, KEYFROM(expression) option, and KEYTO(reference) option in the *PL/I Language Reference*.

Relative byte addresses (RBA)

Relative byte addresses allow you to use keyed access on an ESDS associated with a KEYED SEQUENTIAL file.

The RBAs, or keys, are character strings of length 4, and their values are defined by VSAM. You cannot construct or manipulate RBAs in PL/I; you can, however, compare their values in order to determine the relative positions of records within the data set. RBAs are not normally printable.

You can obtain the RBA for a record by using the KEYTO option, either on a WRITE statement when you are loading or extending the data set, or on a READ statement when the data set is being read. You can subsequently use an RBA obtained in either of these ways in the KEY option of a READ or REWRITE statement.

Do not use an RBA in the KEYFROM option of a WRITE statement.

VSAM allows use of the relative byte address as a key to a KSDS, but this use is not supported by PL/I.

Relative record numbers

Records in an RRDS are identified by a relative record number that starts at 1 and increments by 1 for each succeeding record. You can use these relative record numbers as keys for keyed access to the data set.

Keys used as relative record numbers are character strings of length 8. The character value of a source key you use in the KEY or KEYFROM option must represent an unsigned integer. If the source key is not 8 characters long, it is truncated or padded with blanks (interpreted as zeros) on the left. The value returned by the KEYTO option is a character string of length 8, with leading zeros suppressed.

Choosing a data set type

When you plan your program, you must first decide which type of data set to use. There are three types of VSAM data sets and five types of non-VSAM data sets available to you.

VSAM data sets can provide all the function of the other types of data sets, plus additional functions available only in VSAM. VSAM can usually match other data set types in performance, and often improve upon it. However, VSAM is more subject to performance degradation through misuse of function.

For a comparison of all eight types of data sets, see Table 16 on page 236; however, many factors in the choice of data set type for a large installation are beyond the scope of this document.

When choosing between the VSAM data set types, you should base your choice on the most common sequence in which you will require your data. You can use the following procedure to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and how it will be accessed.
   a. Primarily sequentially — favors ESDS.
   b. Primarily by key — favors KSDS.
   c. Primarily by number — favors RRDS.
2. Determine how you will load the data set. Note that you must load a KSDS in key sequence; thus an ESDS with an *alternate index* path can be a more practical alternative for some applications.
3. Determine whether you require access through an *alternate index* path. These are only supported on KSDS and ESDS. If you require an *alternate index* path, determine whether the *alternate index* will have unique or nonunique keys. Use of nonunique keys can limit key processing. However, it might also be impractical to assume that you will use unique keys for all future records; if you attempt to insert a record with a nonunique key in an index that you have created for unique keys, it will cause an error.
4. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported. Table 29 might be helpful.

Do not try to access a dummy VSAM data set, because you will receive an error message indicating that you have an undefined file.

Table 29 shows the compatible file attribute combinations based on the type of VSAM data sets.

Table 29. VSAM data sets and allowed file attributes

	SEQUENTIAL	**KEYED SEQUENTIAL**	**DIRECT**
INPUT	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	
OUTPUT	ESDS	ESDS	KSDS
	RRDS	KSDS	RRDS
		RRDS	Path(U)
UPDATE	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	

Table 29. VSAM data sets and allowed file attributes (continued)

	SEQUENTIAL	KEYED SEQUENTIAL	DIRECT

Key:

ESDS Entry-sequenced data set

KSDS Key-sequenced data set

RRDS Relative record data set

Path(N)
Alternate index path with nonunique keys

Path(U)
Alternate index path with unique keys

Notes:

- You can combine the attributes on the left with those at the top of the figure for the data sets and paths shown. For example, only an ESDS and an RRDS can be SEQUENTIAL OUTPUT.
- PL/I does not support dummy VSAM data sets.

Table 30. Processing allowed on alternate index paths

Base cluster type	Alternate index key type	Processing	Restrictions
KSDS	Unique key	As normal KSDS	Cannot modify key of access.
	Nonunique key	Limited keyed access	Cannot modify key of access.
ESDS	Unique key	As KSDS	No deletion.
			Cannot modify key of access.
	Nonunique key	Limited keyed access	No deletion.
			Cannot modify key of access.

Related information:

"Entry-sequenced data sets" on page 302
This topic describes the statements and options that are allowed for files associated with an entry-sequenced data set (ESDS).

"Key-sequenced and indexed entry-sequenced data sets" on page 306
An indexed data set can be a key-sequenced data set (KSDS) with its prime index; it can also be a KSDS or an entry-sequenced data set (ESDS) with an *alternate index*. This topic describes the statements and options that are allowed for files associated with indexed VSAM data sets.

"Relative-record data sets" on page 320
This topic describes the statements and options that are allowed for files associated with VSAM relative-record data sets (RRDS).

Defining files for VSAM data sets

This topic describes the file declaration that you can use to define a sequential or direct VSAM data set.

Defining sequential VSAM data set

You define a sequential VSAM data set by using a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

Defining direct VSAM data set

You define a direct VSAM data set by using a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
            [KEYED]
             ENVIRONMENT(options);
```

Table 15 on page 228 shows the default attributes. The file attributes are described in the *PL/I Language Reference*. For information about the options of the ENVIRONMENT attribute, see "Specifying ENVIRONMENT options."

Some combinations of the file attributes INPUT, OUTPUT, or UPDATE and DIRECT, SEQUENTIAL, or KEYED SEQUENTIAL are allowed only for certain types of VSAM data sets. Table 29 on page 296 shows the compatible combinations.

Specifying ENVIRONMENT options

This section describes the ENVIRONMENT options that are applicable to VSAM data sets.

Many of the options of the ENVIRONMENT attribute affecting data set structure are not needed for VSAM data sets. If you specify them, they are either ignored or used for checking purposes. If those that are checked conflict with the values defined for the data set, the UNDEFINEDFILE condition is raised when an attempt is made to open the file.

You can use the following ENVIRONMENT options for VSAM data sets:

```
BKWD
BUFND (n)
BUFNI (n)
BUFSP (n)
GENKEY
PASSWORD (password-specification)
REUSE
SCALARVARYING
SKIP
VSAM
```

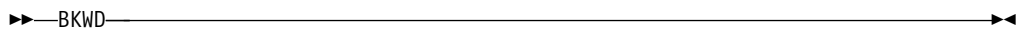
GENKEY and SCALARVARYING options have the same effect as they do when you use them for non-VSAM data sets. Note that under VSAM RLS, options BUFND, BUFNI, and BUFSP are ignored.

The options that are checked for a VSAM data set are RECSIZE and, for a key-sequenced data set, KEYLENGTH and KEYLOC. Table 15 on page 228 shows which options are ignored for VSAM. Table 15 on page 228 also shows the required and default options.

For VSAM data sets, you specify the maximum and average lengths of the records to the Access Method Services utility when you define the data set. If you include the RECSIZE option in the file declaration for checking purposes, specify the maximum record size. If the RECSIZE value that you specify conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

BKWD

The BKWD option specifies backward processing for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file associated with a VSAM data set.



Sequential reads (that is, reads without the KEY option) retrieve the previous record in sequence. For indexed data sets, the previous record is, in general, the record with the next lower key. However, if you are accessing the data set through a nonunique *alternate index*, records with the same key are recovered in their normal sequence. For example, consider the following records, where C1, C2, and C3® have the same key:

```
A B C1 C2 C3 D E
```

The records are recovered in the following sequence:

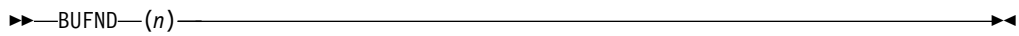
```
E D C1 C2 C3 B A
```

When a file with the BKWD option is opened, the data set is positioned at the last record. ENDFILE is raised in the normal way when the start of the data set is reached.

Do not specify the BKWD option with either the REUSE option or the GENKEY option. Also, the WRITE statement is not allowed for files declared with the BKWD option.

BUFND

The BUFND option specifies the number of data buffers required for a VSAM data set.

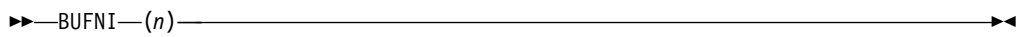


n Specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

Multiple data buffers help performance when the file has the SEQUENTIAL attribute and you are processing long group of contiguous records sequentially.

BUFNI

The BUFNI option specifies the number of index buffers required for a VSAM key-sequence data set.



n Specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

Multiple index buffers help performance when the file has the KEYED attribute. Specify at least as many index buffers as there are levels in the index.

BUFSP

The BUFSP option specifies, in bytes, the total buffer space required for a VSAM data set (for both the data and index components).

n Specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

It is usually preferable to specify the BUFNI and BUFND options rather than BUFSP.

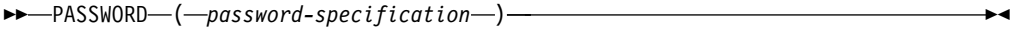
GENKEY

The GENKEY (generic key) option applies only to INDEXED and VSAM key-sequenced data sets. You can use this option to classify keys recorded in a data set and use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

For detailed information, see "GENKEY option — key classification" on page 234.

PASSWORD

When you define a VSAM data set to the system (by using the DEFINE command of Access Method Services), you can associate READ and UPDATE passwords with it. From that point on, you must include the appropriate password in the declaration of any PL/I file that you use to access the data set.



password-specification
Specifies a character constant or character variable that specifies the password for the type of access your program requires. If you specify a constant, it must not contain a repetition factor; if you specify a variable, it must be level-1, element, static, and unsubscripted.

The character string is padded or truncated to 8 characters and passed to VSAM for inspection. If the password is incorrect, the system operator is given a number of chances to specify the correct password. You specify the number of chances to be allowed when you define the data set. After this number of unsuccessful tries, the UNDEFINEDFILE condition is raised.

REUSE

The REUSE option specifies that an OUTPUT file associated with a VSAM data set is to be used as a work file.

The data set is treated as an empty data set each time the file is opened. Any secondary allocations for the data set are released, and the data set is treated exactly as if it were being opened for the first time.

Do not associate a file that has the REUSE option with a data set that has *alternate indexes* or the BKWD option, and do not open it for INPUT or UPDATE.

The REUSE option takes effect only if you specify REUSE in the Access Method Services DEFINE CLUSTER command.

SKIP

The SKIP option specifies that the VSAM OPTCD "SKP" is to be used whenever possible. It is applicable to key-sequenced data sets that you access by means of a KEYED SEQUENTIAL INPUT or UPDATE file.

You should specify this option for the file if your program accesses individual records scattered throughout the data set, but does so primarily in ascending key order.

Omit this option if your program reads large numbers of records sequentially without the use of the KEY option, or if it inserts large numbers of records at specific points in the data set (mass sequential insert).

It is never an error to specify (or omit) the SKIP option; its effect on performance is significant only in the circumstances described.

VSAM

You must specify the VSAM option for VSAM data sets.

Performance options

You can specify the buffer options in the AMP parameter of the DD statement; they are explained in your Access Method Services manual.

Defining files for alternate index paths

VSAM allows you to define alternate indexes on key sequenced and entry sequenced data sets.

Using alternate indexes, you can access key sequenced data sets in a number of ways other than from the prime index; you can also index and access entry sequenced data sets by key or sequentially in order of the keys. Consequently, data created in one form can be accessed in a large number of different ways. For example, an employee file might be indexed by personnel number, by name, and also by department number.

When an alternate index has been built, you actually access the data set through a third object known as an alternate index *path* that acts as a connection between the alternate index and the data set.

Two types of alternate indexes are allowed—unique key and nonunique key. For a unique key alternate index, each record must have a different alternate key. For a nonunique key alternate index, any number of records can have the same alternate key. In the example suggested above, the alternate index using the names can be a unique key alternate index (provided each person had a different name). The alternate index using the department number would be a nonunique key alternate index because more than one person would be in each department.

In most respects, you can treat a data set accessed through a unique key alternate index path like a KSDS accessed through its prime index. You can access the records by key or sequentially, you can update records, and you can add new records. If the data set is a KSDS, you can delete records, and alter the length of updated records. Restrictions and allowed processing are shown in Table 30 on page 297. When you add or delete records, all indexes associated with the data set are by default altered to reflect the new situation.

In data sets accessed through a nonunique key alternate index path, the record accessed is determined by the key and the sequence. The key can be used to establish positioning so that sequential access can follow. The use of the key accesses the first record with that key. When the data set is read backwards, only the order of the keys is reversed. The order of the records with the same key remains the same whichever way the data set is read.

Defining VSAM data sets

You can use the DEFINE CLUSTER command of Access Method Services to define and catalog VSAM data sets.

To use the DEFINE command, you need to know the following information:

- The name and password of the master catalog if the master catalog is password protected
- The name and password of the VSAM private catalog you are using if you are not using the master catalog
- Whether VSAM space for your data set is available
- The type of VSAM data set you are going to create
- The volume on which your data set is to be placed
- The average and maximum record size in your data set
- The position and length of the key for an indexed data set
- The space to be allocated for your data set
- How to code the DEFINE command
- How to use the Access Method Services program

When you have the information, you can code the DEFINE command and then define and catalog the data set by using Access Method Services.

Entry-sequenced data sets

This topic describes the statements and options that are allowed for files associated with an entry-sequenced data set (ESDS).

Table 31. Statements and options allowed for loading and accessing VSAM entry-sequenced data sets

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)

Table 31. Statements and options allowed for loading and accessing VSAM entry-sequenced data sets (continued)

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference);	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference)[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)[3]

Notes:

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use either of the options KEY or KEYTO, it must also include the attribute KEYED.
2. The statement `READ FILE(file-reference);` is equivalent to the statement `READ FILE(file-reference) IGNORE (1);`.
3. The expression used in the KEY option must be a relative byte address, previously obtained by the KEYTO option.

Loading an ESDS

When an ESDS is being loaded, the associated file must be opened for SEQUENTIAL OUTPUT. The records are retained in the order in which they are presented.

You can use the KEYTO option to obtain the relative byte address of each record as it is written. You can subsequently use these keys to achieve keyed access to the data set.

Using a SEQUENTIAL file to access an ESDS

You can open a SEQUENTIAL file that is used to access an ESDS with either the INPUT or the UPDATE attribute. If you use either of the options KEY or KEYTO, the file must also have the KEYED attribute.

Sequential access is in the order that the records were originally loaded into the data set. You can use the KEYTO option on the READ statements to recover the RBAs of the records that are read. If you use the KEY option, the record that is recovered is the one with the RBA you specify. Subsequent sequential access continues from the new position in the data set.

For an UPDATE file, the WRITE statement adds a new record at the end of the data set. With a REWRITE statement, the record rewritten is the one with the

specified RBA if you use the KEY option; otherwise, it is the record accessed on the previous READ. You must not attempt to change the length of the record that is being replaced with a REWRITE statement.

The DELETE statement is not allowed for entry-sequenced data sets.

Defining and loading an ESDS

This topic shows an example PL/I program that defines and loads an entry-sequenced data set (ESDS). The program writes to the data set by using a SEQUENTIAL OUTPUT file.

In the program shown in Figure 41 on page 305, the data set is defined with the DEFINE CLUSTER command and given the name PLIVSAM.AJC1.BASE. The NONINDEXED keyword causes an ESDS to be defined.

The PL/I program writes to the data set by using a SEQUENTIAL OUTPUT file and a WRITE FROM statement. The DD statement for the file contains the DSNAME of the data set given in the NAME parameter of the DEFINE CLUSTER command.

You can obtain the RBA of the records as keys in a KEYED file for subsequent use. To do this, you must declare a suitable variable to hold the key and use the WRITE...KEYTO statement. See the following example:

```
DCL CHARS CHAR(4);
WRITE FILE(FAMFILE) FROM (STRING)
 KEYTO(CHARS);
```

Note that the keys would not normally be printable, but could be retained for subsequent use.

The cataloged procedure IBMZCBG is used. Because the same program (in Figure 41 on page 305) can be used for adding records to the data set, it is retained in a library. For more information about the procedure of adding records, see the example in “Updating an ESDS” on page 305.

```
//OPT9#7  JOB
//STEP1    EXEC   PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD    *
     DEFINE CLUSTER -
      (NAME(PLIVSAM.AJC1.BASE) -
      VOLUMES(nnnnnn) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2))
/*
//STEP2  EXEC IBMZCLG
//PLI.SYSIN      DD *
   CREATE: PROC OPTIONS(MAIN);

      DCL
        FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
        IN FILE RECORD INPUT,
        STRING CHAR(80),
        EOF BIT(1) INIT('0'B);

      ON ENDFILE(IN) EOF='1'B;

      READ FILE(IN) INTO (STRING);
      DO I=1 BY 1 WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
        WRITE FILE(FAMFILE) FROM (STRING);
        READ FILE(IN) INTO (STRING);
      END;

      PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
    END;
/*
//LKED.SYSLMOD  DD  DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
//         UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.FAMFILE DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=OLD
//GO.IN      DD     *
FRED                     69          M
ANDY                     70          M
SUZAN                    72          F
/*
```

Figure 41. Defining and loading an ESDS

Updating an ESDS

This topic shows an example of adding a new record at the end of an ESDS.

This example is based on the PL/I program shown in Figure 41. That program can also be used for adding records to the data set. In the program, the data set PLIVSAM.AJC1.BASE is defined with the DEFINE CLUSTER command.

Figure 42 on page 306 shows the addition of a new record at the end of the ESDS. A SEQUENTIAL OUTPUT file is used, and the data set associated with it, PLIVSAM.AJC1.BASE, is specified in the DSNAME parameter.

```
//OPT9#8   JOB
//STEP1    EXEC  PGM=PGMA
//STEPLIB   DD   DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP)
//          DD   DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//FAMFILE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//IN        DD   *
JANE                    75          F
//
```

Figure 42. Updating an ESDS

You can rewrite existing records in an ESDS, provided that the length of the record is not changed. You can use a SEQUENTIAL or KEYED SEQUENTIAL update file to do this. If you use keys, they can be the RBAs or keys of an *alternate index* path.

Delete statements are not allowed for ESDS.

Key-sequenced and indexed entry-sequenced data sets

An indexed data set can be a key-sequenced data set (KSDS) with its prime index; it can also be a KSDS or an entry-sequenced data set (ESDS) with an *alternate index*. This topic describes the statements and options that are allowed for files associated with indexed VSAM data sets.

Except where otherwise stated, the description in Table 32 applies to all indexed VSAM data sets.

Table 32. Statements and options allowed for loading and accessing VSAM indexed data sets

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)

Table 32. Statements and options allowed for loading and accessing VSAM indexed data sets (continued)

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference)	KEY(expression)
DIRECT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	DELETE FILE(file-reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Table 32. Statements and options allowed for loading and accessing VSAM indexed data sets (continued)

File declaration[1]	Valid statements with options you must include	Other options you can also include

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the declaration.
2. The statement `READ FILE(file-reference);` is equivalent to the statement `READ FILE(file-reference) IGNORE(1);`.
3. Do not associate a SEQUENTIAL OUTPUT file with a data set accessed through an *alternate index*.
4. Do not associate a DIRECT file with a data set accessed through a nonunique *alternate index*.
5. DELETE statements are not allowed for a file associated with an ESDS accessed through an *alternate index*.

Loading a KSDS or indexed ESDS

When a KSDS is being loaded, you must open the associated file for KEYED SEQUENTIAL OUTPUT. You must present the records in ascending key order, and you must use the KEYFROM option.

Note that you must use the prime index for loading the data set; you cannot load a VSAM data set through an *alternate index*.

If a KSDS already contains some records and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can add records only at the end of the data set. The rules given in the previous paragraph apply; in particular, the first record you present must have a key greater than the highest key present on the data set.

Figure 43 on page 309 shows the DEFINE command used to define a KSDS. The data set is given the name PLIVSAM.AJC2.BASE and defined as a KSDS because of the use of the INDEXED operand. The position of the keys within the record is defined in the KEYS operand.

Within the PL/I program, a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A KSDS must be loaded in this manner.

The file is associated with the data set by a DD statement that uses the name given in the DEFINE command as the DSNAME parameter.

```
//OPT9#12  JOB
//   EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT  DD  SYSOUT=A
//SYSIN     DD  *
   DEFINE CLUSTER -
     (NAME(PLIVSAM.AJC2.BASE) -
     VOLUMES(nnnnnn) -
     INDEXED -
     TRACKS(3 1) -
     KEYS(20 0) -
     RECORDSIZE(23 80))
/*
//   EXEC IBMZCBG
//PLI.SYSIN      DD *
 TELNOS: PROC OPTIONS(MAIN);

        DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD POS(1),
            NUMBER CHAR(3) DEF CARD POS(21),
            OUTREC CHAR(23) DEF CARD POS(1),
            EOF BIT(1) INIT('0'B);

        ON ENDFILE(SYSIN) EOF='1'B;

        OPEN FILE(DIREC) OUTPUT;

        GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
        WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;

       CLOSE FILE(DIREC);

        END TELNOS;
/*
//GO.DIREC  DD  DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN  DD  *
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,D.        141
CORY,G.             336
ELLIOTT,D.          875
FIGGINS,S.          413
HARVEY,C.D.W.       205
HASTINGS,G.M.       391
KENDALL,J.G.        294
LANCASTER,W.R.      624
MILES,R.            233
NEWMAN,M.W.         450
PITT,W.H.           515
ROLF,D.E.           114
SHEERS,C.D.         241
SUTCLIFFE,M.        472
TAYLOR,G.C.         407
WILTON,L.W.         404
WINSTONE,E.M.       307
//
```

Figure 43. Defining and loading a key-sequenced data set (KSDS)

Using a SEQUENTIAL file to access a KSDS or indexed ESDS

You can open a SEQUENTIAL file that is used to access a KSDS with either the INPUT or the UPDATE attribute.

For READ statements without the KEY option, the records are recovered in ascending key order (or in descending key order if the BKWD option is used). You can obtain the key of a record recovered in this way by using the KEYTO option.

If you use the KEY option, the record recovered by a READ statement is the one with the specified key. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, without respect to the position of any previous access. If you are accessing the data set through a unique index, the KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists on the data set. For a nonunique index, subsequent retrieval of records with the same key is in the order that they were added to the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the first record with the specified key; otherwise, it is the record that was accessed by the previous READ statement. When you rewrite a record by using an *alternate index*, do not change the prime key of the record.

Using a DIRECT file to access a KSDS or indexed ESDS

You can open a DIRECT file that is used to access an indexed VSAM data set with the INPUT, OUTPUT, or UPDATE attribute. Do not use a DIRECT file to access the data set through a nonunique index.

If you use a DIRECT OUTPUT file to add records to the data set, and if an attempt is made to insert a record with the same key as a record that already exists, the KEY condition is raised.

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

Figure 44 on page 311 shows one method to update a KSDS by using the prime index.

```
//OPT9#13  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 DIRUPDT: PROC OPTIONS(MAIN);

         DCL DIREC FILE RECORD KEYED ENV(VSAM),
             ONCODE BUILTIN,
             OUTREC CHAR(23),
             NUMBER CHAR(3) DEF OUTREC POS(21),
             NAME CHAR(20) DEF OUTREC,
             CODE CHAR(1),
             EOF BIT(1) INIT('0'B);

         ON ENDFILE(SYSIN) EOF='1'B;

         ON KEY(DIREC) BEGIN;
          IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                            ('NOT FOUND: ',NAME)(A(15),A);
          IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                           ('DUPLICATE: ',NAME)(A(15),A);
          END;

         OPEN FILE(DIREC) DIRECT UPDATE;

       GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
         (COLUMN(1),A(20),A(3),A(1));
       DO WHILE (¬EOF);
       PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
       (A(1),A(20),A(1),A(3),A(1),A(1));
       SELECT (CODE);
          WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
          WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
          WHEN('D') DELETE FILE(DIREC) KEY(NAME);
          OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME) (A(15),A);
       END;
       GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
         (COLUMN(1),A(20),A(3),A(1));
       END;

         CLOSE FILE(DIREC);
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(DIREC) SEQUENTIAL INPUT;

         EOF='0'B;
         ON ENDFILE(DIREC) EOF='1'B;

         READ FILE(DIREC) INTO(OUTREC);
         DO WHILE(¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
         READ FILE(DIREC) INTO(OUTREC);
         END;
         CLOSE FILE(DIREC);
   END DIRUPDT;
```

Figure 44. Updating a KSDS

```
/*
//GO.DIREC DD DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.         516C
GOODFELLOW,D.T.     889A
MILES,R.               D
HARVEY,C.D.W.       209A
BARTLETT,S.G.       183A
CORY,G.                D
READ,K.M.           001A
PITT,W.H.
ROLF,D.F.              D
ELLIOTT,D.          291C
HASTINGS,G.M.          D
BRAMLEY,O.H.        439C
/*
```

Updating a KSDS (continued)

A DIRECT update file is used and the data is altered according to a code that is passed in the records in the file SYSIN:

A Add a new record
C Change the number of an existing name
D Delete a record

At the label NEXT, the name, number, and code are read in and action taken according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished (at the label PRINT), the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

The file is associated with the data set by a DD statement that uses the DSNAME PLIVSAM.AJC2.BASE defined in the Access Method Services DEFINE CLUSTER command in Figure 43 on page 309.

Updating a KSDS

There are a number of methods of updating a KSDS. For mass sequential insertion, use a KEYED SEQUENTIAL UPDATE file.

This method gives faster performance because the data is written onto the data set only when strictly necessary and not after every write statement, and also because the balance of free space within the data set is retained.

You can use the following statements to achieve effective mass sequential insertion:

```
DCL DIREC KEYED SEQUENTIAL UPDATE
    ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
 KEYFROM(NAME);
```

The PL/I input/output routines detect that the keys are in sequence and make the correct requests to VSAM. If the keys are not in sequence, this too is detected and no error occurs, although the performance advantage is lost.

The example shown in Figure 44 on page 311 uses a DIRECT file to update a KSDS. This method is suitable for the data as is shown in the example.

Alternate indexes for KSDSs or indexed ESDSs

Alternate indexes allow you to access KSDSs or indexed ESDSs in various ways, by using either unique or nonunique keys.

Creating unique key alternate index path for ESDS

This topic provides an example that illustrates how to create a unique key alternate index path for an ESDS.

Figure 45 shows the creation of a unique key alternate index path for the ESDS defined and loaded in Figure 41 on page 305. Using this path, the data set is indexed by the name of the child in the first 15 bytes of the record.

The following Access Method Services commands are used:

DEFINE ALTERNATEINDEX
Defines the alternate index as a data set to VSAM.

BLDINDEX
Places the pointers to the relevant records in the alternate index.

DEFINE PATH
Defines an entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE operands of BLDINDEX and for the sort files. Ensure that the correct names are specified at the various points.

```
//OPT9#9    JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC1.ALPHIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 1) -
     KEYS(15 0) -
     RECORDSIZE(20 40) -
     UNIQUEKEY -
    RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC1.ALPHIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC1.ALPHPATH) -
     PATHENTRY(PLIVSAM.AJC1.ALPHIND))
//
```

Figure 45. Creating a unique key alternate index path for an ESDS

Creating nonunique key alternate index path for ESDS

This topic provides an example that illustrates how to create a nonunique key alternate index path for an ESDS.

Figure 46 on page 314 shows the creation of a nonunique key alternate index path for an ESDS. The alternate index enables the data to be selected by the gender of

the children. This enables the girls or the boys to be accessed separately and every member of each group to be accessed by use of the key.

The following Access Method Services commands are used:

DEFINE ALTERNATEINDEX
Defines the alternate index as a data set to VSAM.

BLDINDEX
Places the pointers to the relevant records in the alternate index.

DEFINE PATH
Defines an entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE operands of BLDINDEX and for the sort files. Ensure that the correct names are specified at the various points.

In this example, the NONUNIQUEKEY operand specifies that the index has nonunique keys. When creating an index with nonunique keys, ensure that the RECORDSIZE you specify is large enough. In a nonunique alternate index, each alternate index record contains pointers to all the records that have the associated index key. The pointer takes the form of an RBA for an ESDS and the prime key for a KSDS. When a large number of records might have the same key, a large record is required.

```
//OPT9#10   JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   /* care must be taken with recordsize */
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC1.SEXIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 1) -
     KEYS(1 37) -
     RECORDSIZE(20 400) -
     NONUNIQUEKEY -
    RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC1.SEXIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC1.SEXPATH) -
     PATHENTRY(PLIVSAM.AJC1.SEXIND))
//
```

Figure 46. Creating a nonunique key alternate index path for an ESDS

Creating unique key alternate index path for KSDS

This topic provides an example that illustrates how to create a unique key alternate index path for a KSDS.

Figure 47 on page 315 shows the creation of a unique key alternate index path for a KSDS. The data set is indexed by the telephone number, enabling the number to

be used as a key to discover the name of the person on that extension. Also, the data set can be listed in numerical order to show which numbers are not used.

In this example, the UNIQUEKEY operand specifies that keys are unique.

The following Access Method Services commands are used:

DEFINE ALTERNATEINDEX
: Defines the data set that will hold the alternate index data.

BLDINDEX
: Places the pointer to the relevant records in the alternate index.

DEFINE PATH
: Defines the entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE of BLDINDEX and for the sort files. Be careful not to confuse the names involved.

```
//OPT9#14   JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC2.NUMIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 4) -
     KEYS(3 20) -
     RECORDSIZE(24 48) -
     UNIQUEKEY -
    RELATE(PLIVSAM.AJC2.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC2.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC2.NUMIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC2.NUMPATH) -
     PATHENTRY(PLIVSAM.AJC2.NUMIND))
//
```

Figure 47. Creating a unique key alternate index path for a KSDS

When creating an alternate index with a unique key, ensure that no further records could be included with the same alternate key. In practice, a unique key alternate index would not be entirely satisfactory for a telephone directory as it would not allow two people to have the same number. Similarly, the prime key would prevent one person having two numbers. A solution would be to have an ESDS with two nonunique key alternate indexes, or to restructure the data format to allow more than one number per person and to have a nonunique key alternate index for the numbers.

Detecting nonunique alternate index keys

If you are accessing a VSAM data set through an alternate index path, the presence of nonunique keys can be detected by the SAMEKEY built-in function.

After each retrieval, SAMEKEY indicates whether any further records exist with the same alternate index key as the record just retrieved. Hence, it is possible to stop at the last of a series of records with nonunique keys without having to read beyond the last record. SAMEKEY (file-reference) returns `'1'B` if the input/output statement has completed successfully and the accessed record is followed by another with the same key; otherwise, it returns `'0'B`.

Using alternate indexes with ESDSs

Figure 48 on page 317 shows the use of alternate indexes and backward reading on an ESDS. The program has four files:

BASEFLE
Reads the base data set forward.

BACKFLE
Reads the base data set backward.

ALPHFLE
Is the alphabetic alternate index path indexing the children by name.

SEXFILE
Is the alternate index path that corresponds to the gender of the children.

There are DD statements for all the files. They connect BASEFLE and BACKFLE to the base data set by specifying the name of the base data set in the DSNAME parameter, and connect ALPHFLE and SEXFILE by specifying the names of the paths given in Figure 45 on page 313 and Figure 46 on page 314.

The program uses SEQUENTIAL files to access the data and print it first in the normal order, then in the reverse order. At the label AGEQUERY, a DIRECT file is used to read the data associated with an alternate index key in the unique alternate index.

Finally, at the label SPRINT, a KEYED SEQUENTIAL file is used to print a list of the females in the family, using the nonunique key alternate index path. The SAMEKEY built-in function is used to read all the records with the same key. The names of the females will be accessed in the order in which they were originally entered. This will happen whether the file is read forward or backward. For a nonunique key path, the BKWD option only affects the order in which the keys are read; the order of items with the same key remains the same as it is when the file is read forward.

At the end of the example, the Access Method Services DELETE command is used to delete the base data set. When this is done, the associated alternate indexes and paths will also be deleted.

Using alternate indexes with KSDSs

Figure 49 on page 319 shows the use of a path with a unique alternate index key to update a KSDS and then to access and print it in the order of the alternate index.

The alternate index path is associated with the PL/I file by a DD statement that specifies the name of the path (PLIVSAM.AJC2.NUMPATH, given in the DEFINE PATH command in Figure 47 on page 315) as the DSNAME.

In the first section of the program, a DIRECT OUTPUT file is used to insert a new record by using the alternate index key. Note that any alteration made with an

alternate index must not alter the prime key or the alternate index key of access of an existing record. Also, the alternation must not add a duplicate key in the prime index or in any unique key alternate index.

In the second section of the program (at the label PRINTIT), the data set is read in the order of the alternate index keys by using a SEQUENTIAL INPUT file. It is then printed onto SYSPRINT.

```
//OPT9#15  JOB
//STEP1  EXEC IBMZCLG
//PLI.SYSIN      DD *
  READIT: PROC OPTIONS(MAIN);
    DCL   BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
                 /*File to read base data set forward */
          BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
                 /*File to read base data set backward */
          ALPHFLE FILE DIRECT INPUT ENV(VSAM),
          /*File to access via unique alternate index path */
          SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
           /*File to access via nonunique alternate index path */
          STRING CHAR(80),  /*String to be read into */
          1 STRUC DEF (STRING),
            2 NAME CHAR(25),
            2 DATE_OF_BIRTH CHAR(2),
            2 FILL CHAR(10),
            2 SEX CHAR(1);
            DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;
            DCL EOF BIT(1) INIT('0'B);

       /*Print out the family eldest first*/

       ON ENDFILE(BASEFLE) EOF='1'B;
       PUT EDIT('FAMILY ELDEST FIRST')(A);
       READ FILE(BASEFLE) INTO (STRING);
       DO WHILE(¬EOF);
          PUT SKIP EDIT(STRING)(A);
          READ FILE(BASEFLE) INTO (STRING);
       END;
       CLOSE FILE(BASEFLE);
       PUT SKIP(2);
        /*Close before using data set from other file not
          necessary but good practice to prevent potential
          problems*/

       EOF='0'B;
       ON ENDFILE(BACKFLE) EOF='1'B;
       PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);
       READ FILE(BACKFLE) INTO(STRING);
       DO WHILE(¬EOF);
         PUT SKIP EDIT(STRING)(A);
         READ FILE(BACKFLE) INTO (STRING);
       END;

       CLOSE FILE(BACKFLE);
       PUT SKIP(2);

       /*Print date of birth of child specified in the file
         SYSIN*/
       ON KEY(ALPHFLE) BEGIN;
          PUT SKIP EDIT
             (NAMEHOLD,' NOT A MEMBER OF THE SMITH FAMILY') (A);
          GO TO SPRINT;
       END;
```

Figure 48. Alternate index paths and backward reading with an ESDS

```
 AGEQUERY:
    EOF='0'B;
    ON ENDFILE(SYSIN) EOF='1'B;
    GET SKIP EDIT(NAMEHOLD)(A(25));
    DO WHILE(¬EOF);
          READ FILE(ALPHFLE) INTO (STRING) KEY(NAMEHOLD);
          PUT SKIP (2) EDIT(NAMEHOLD,' WAS BORN IN ',
              DATE_OF_BIRTH)(A,X(1),A,X(1),A);
          GET SKIP EDIT(NAMEHOLD)(A(25));
    END;
 SPRINT:
    CLOSE FILE(ALPHFLE);
    PUT SKIP(1);

   /*Use the alternate index to print out all the females in the
     family*/
       ON ENDFILE(SEXFILE) GOTO FINITO;
       PUT SKIP(2) EDIT('ALL THE FEMALES')(A);
       READ FILE(SEXFILE) INTO (STRING) KEY('F');
       PUT SKIP EDIT(STRING)(A);
       DO WHILE(SAMEKEY(SEXFILE));
          READ FILE(SEXFILE) INTO (STRING);
          PUT SKIP EDIT(STRING)(A);
       END;

 FINITO:
     END;
/*
//GO.BASEFLE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//GO.BACKFLE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//GO.ALPHFLE   DD   DSN=PLIVSAM.AJC1.ALPHPATH,DISP=SHR
//GO.SEXFILE   DD   DSN=PLIVSAM.AJC1.SEXPATH,DISP=SHR
//GO.SYSIN     DD    *
ANDY
/*
//STEP2    EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT    DD    SYSOUT=A
//SYSIN       DD     *
     DELETE -
          PLIVSAM.AJC1.BASE
//
```

Alternate Index Paths and Backward Reading with an ESDS (continued)

```
//OPT9#16   JOB
//STEP1     EXEC IBMZCLG,REGION.GO=256K
//PLI.SYSIN DD   *
   ALTER: PROC OPTIONS(MAIN);
     DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
         NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
         IN FILE RECORD,
         STRING CHAR(80),
         NAME CHAR(20) DEF STRING,
         NUMBER CHAR(3) DEF STRING POS(21),
         DATA CHAR(23) DEF STRING,
         EOF BIT(1) INIT('0'B);

     ON KEY (NUMFLE1) BEGIN;
       PUT SKIP EDIT('DUPLICATE NUMBER')(A);
     END;

     ON ENDFILE(IN) EOF='1'B;

     READ FILE(IN) INTO (STRING);
     DO WHILE(¬EOF);
       PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
       WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
       READ FILE(IN) INTO (STRING);
     END;

     CLOSE FILE(NUMFLE1);

     EOF='0'B;
     ON ENDFILE(NUMFLE2) EOF='1'B;

     READ FILE(NUMFLE2) INTO (STRING);
     DO WHILE(¬EOF);
       PUT SKIP EDIT(DATA)(A);
       READ FILE(NUMFLE2) INTO (STRING);
     END;

     PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
  END;
/*
//GO.IN    DD   *
RIERA L            123
/*
//NUMFLE1    DD   DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//NUMFLE2    DD   DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//STEP2    EXEC PGM=IDCAMS,COND=EVEN
//SYSPRINT   DD  SYSOUT=A
//SYSIN    DD    *
     DELETE -
     PLIVSAM.AJC2.BASE
//
```

Figure 49. Using a unique alternate index path to access a KSDS

Relative-record data sets

This topic describes the statements and options that are allowed for files associated with VSAM relative-record data sets (RRDS).

Table 33. Statements and options allowed for loading and accessing VSAM relative-record data sets

File declaration[1]	Valid statements with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	

Table 33. Statements and options allowed for loading and accessing VSAM relative-record data sets (continued)

File declaration[1]	Valid statements with options you must include	Other options you can also include
DIRECT UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); READ FILE(file-reference) SET(pointer-reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); DELETE FILE(file-reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, your declaration must also include the attribute KEYED.

   The UNLOCK statement for DIRECT UPDATE files is ignored if you use it for files associated with a VSAM RRDS.
2. The statement `READ FILE(file-reference);` is equivalent to the statement `READ FILE(file-reference) IGNORE(1);`.

Loading an RRDS

When an RRDS is being loaded, you must open the associated file for OUTPUT. Use either a DIRECT or a SEQUENTIAL file.

For a DIRECT OUTPUT file, each record is placed in the position specified by the relative record number (or key) in the KEYFROM option of the WRITE statement (see "Keys for VSAM data sets" on page 294).

For a SEQUENTIAL OUTPUT file, use WRITE statements with or without the KEYFROM option. If you specify the KEYFROM option, the record is placed in the specified slot; if you omit it, the record is placed in the slot following the current position. There is no requirement for the records to be presented in ascending relative record number order. If you omit the KEYFROM option, you can obtain the relative record number of the written record by using the KEYTO option.

If you want to load an RRDS sequentially, without use of the KEYFROM or KEYTO options, your file is not required to have the KEYED attribute.

It is an error to attempt to load a record into a position that already contains a record: if you use the KEYFROM option, the KEY condition is raised; if you omit it, the ERROR condition is raised.

In Figure 50 on page 323, the data set is defined with a DEFINE CLUSTER command and given the name PLIVSAM.AJC3.BASE. The fact that it is an RRDS is determined by the NUMBERED keyword. In the PL/I program, it is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

If the data had been in order and the keys in sequence, it would have been possible to use a SEQUENTIAL file and write into the data set from the start. The records would then have been placed in the next available slot and given the appropriate number. The number of the key for each record could have been returned by using the KEYTO option.

The PL/I file is associated with the data set by the DD statement, which uses as the DSNAME the name given in the DEFINE CLUSTER command.

```
//OPT9#17  JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD *
         DEFINE CLUSTER -
             (NAME(PLIVSAM.AJC3.BASE) -
             VOLUMES(nnnnnn) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20))
/*
//STEP2  EXEC IBMZCBG
//PLI.SYSIN      DD *
 CRR1:   PROC OPTIONS(MAIN);
         DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
             CARD CHAR(80),
             NAME CHAR(20) DEF CARD,
             NUMBER CHAR(2) DEF CARD POS(21),
             IOFIELD CHAR(20),
             EOF BIT(1) INIT('0'B);
         ON ENDFILE (SYSIN) EOF='1'B;
         OPEN FILE(NOS);
         GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         IOFIELD=NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         GET FILE(SYSIN) EDIT(CARD)(A(80));
         END;
         CLOSE FILE(NOS);
 END CRR1;
/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.           12
BAKER,R.            13
BRAMLEY,O.H.        28
CHEESNAME,L.        11
CORY,G.             36
ELLIOTT,D.          85
FIGGINS.E.S.        43
HARVEY,C.D.W.       25
HASTINGS,G.M.       31
KENDALL,J.G.        24
LANCASTER,W.R.      64
MILES,R.            23
NEWMAN,M.W.         40
PITT,W.H.           55
ROLF,D.E.           14
SHEERS,C.D.         21
SURCLIFFE,M.        42
TAYLOR,G.C.         47
WILTON,L.W.         44
WINSTONE,E.M.       37
//
```

Figure 50. Defining and loading a relative-record data set (RRDS)

Using a SEQUENTIAL file to access an RRDS

You can open a SEQUENTIAL file that is used to access an RRDS with either the INPUT or the UPDATE attribute. If you use any of the options KEY, KEYTO, or KEYFROM, your file must also have the KEYED attribute.

For READ statements without the KEY option, the records are recovered in ascending relative record number order. Any empty slots in the data set are skipped.

If you use the KEY option, the record recovered by a READ statement is the one with the relative record number you specify. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with or without the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, regardless of the position of any previous access. For WRITE with the KEYFROM option, the KEY condition is raised if an attempt is made to insert a record with the same relative record number as a record that already exists on the data set. If you omit the KEYFROM option, an attempt is made to write the record in the next slot, relative to the current position. The ERROR condition is raised if this slot is not empty.

You can use the KEYTO option to recover the key of a record that is added by means of a WRITE statement without the KEYFROM option.

REWRITE statements, with or without the KEY option, are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the record with the relative record number you specify; otherwise, it is the record that was accessed by the previous READ statement.

You can use DELETE statements, with or without the KEY option, to delete records from the dataset.

Using a DIRECT file to access an RRDS

A DIRECT file used to access an RRDS can have the OUTPUT, INPUT, or UPDATE attribute. You can read, write, rewrite, or delete records exactly as though a KEYED SEQUENTIAL file were used.

Figure 51 on page 325 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under VSAM.

In the second half of the program, starting at the label PRINT, the updated file is printed out. Again there is no need to check for the empty records as there is in REGIONAL(1).

The PL/I file is associated with the data sets by a DD statement that specifies the DSNAME PLIVSAM.AJC3.BASE, the name given in the DEFINE CLUSTER command in Figure 51 on page 325.

At the end of the example, the DELETE command is used to delete the data set.

```
//* NOTE: WITH A WRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A DUPLICATE MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO CORRESPONDS TO AN EXISTING NUMBER IN THE LIST,
//*       FOR EXAMPLE, ELLIOT.
//*       WITH A REWRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A NOT FOUND MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO DOES NOT CORRESPOND TO AN EXISTING NUMBER IN
//*       THE LIST, FOR EXAMPLE, NEWMAN AND BRAMLEY.
//OPT9#18  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 ACR1:  PROC OPTIONS(MAIN);
                DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
                  (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
                  BYTE CHAR(1) DEF IOFIELD, EOF BIT(1) INIT('0'B),
                  ONCODE BUILTIN;
        ON ENDFILE(SYSIN) EOF='1'B;
        OPEN FILE(NOS) DIRECT UPDATE;
        ON KEY(NOS) BEGIN;
           IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                    ('NOT FOUND:',NAME)(A(15),A);
           IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                    ('DUPLICATE:',NAME)(A(15),A);
        END;
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
           (A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
        SELECT(CODE);
           WHEN('A') WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           WHEN('C') DO;
            DELETE FILE(NOS) KEY(OLDNO);
            WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           END;
           WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
           OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME)(A(15),A);
        END;
```

Figure 51. Updating an RRDS

```
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        END;
         CLOSE FILE(NOS);
            PRINT:
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(NOS) SEQUENTIAL INPUT;
         EOF='0'B;
         ON ENDFILE(NOS) EOF='1'B;
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          DO WHILE (¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD)(A(5),A);
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          END;
       CLOSE FILE(NOS);
 END ACR1;

 /*
 //GO.NOS     DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
 //GO.SYSIN   DD  *
 NEWMAN,M.W.         5640C
 GOODFELLOW,D.T.     89  A
 MILES,R.              23D
 HARVEY,C.D.W.       29  A
 BARTLETT,S.G.       13  A
 CORY,G.               36D
 READ,K.M.           01  A
 PITT,W.H.             55
 ROLF,D.F.             14D
 ELLIOTT,D.          4285C
 HASTINGS,G.M.         31D
 BRAMLEY,O.H.        4928C
 //STEP3    EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
 //SYSPRINT DD SYSOUT=A
 //SYSIN   DD   *
       DELETE -
           PLIVSAM.AJC3.BASE
 //
```

Updating an RRDS (continued)

Using files defined for non-VSAM data sets

Using shared data sets

PL/I allows cross-region or cross-system sharing of data sets. The support for this type of sharing is provided by VSAM.

For details about the support, see the following DFSMS manuals:

- *DFSMS: Using Data Sets*
- *DFSMS: Access Method Services for Catalogs*

Part 3. Improving your program

© Copyright IBM Corp. 1999, 2017

Chapter 15. Improving performance

Many considerations for improving the speed of your program are independent of the compiler that you use and the platform on which it runs. This chapter, however, identifies those considerations that are unique to the PL/I compiler and the code it generates.

Selecting compiler options for optimal performance

The compiler options you choose can greatly improve the performance of the code generated by the compiler; however, like most performance considerations, there are trade-offs associated with these choices.

Fortunately, you can weigh the trade-offs associated with compiler options without editing your source code because these options can be specified on the command line or in the configuration file.

If you want to avoid details, the least complex way to improve the performance of generated code is to specify the following (nondefault) compiler options:

OPT(2) or OPT(3)
DFT(REORDER)

The following topics describe, in more detail, performance improvements and trade-offs associated with specific compiler options.

OPTIMIZE

You can specify the OPTIMIZE option to improve the speed of your program; otherwise, the compiler makes only basic optimization efforts.

Choosing OPTIMIZE(2) directs the compiler to generate code for better performance. Usually, the resultant code is shorter than when the program is compiled under NOOPTIMIZE. Sometimes, however, a longer sequence of instructions runs faster than a shorter sequence. This occurs, for instance, when a branch table is created for a SELECT statement where the values in the WHEN clauses contain gaps. The increased number of instructions generated is usually offset by the execution of fewer instructions in other places.

Choosing OPTIMIZE(3) directs the compiler to generate even better code. However, when you specify the OPTIMIZE(3) option, the compilation will take longer (and sometimes much, much longer) than when you specify OPTIMIZE(2).

GONUMBER

You can specify the GONUMBER option to generate a statement number table that is used for debugging.

This added information can be extremely helpful when you debug, but including statement number tables increases the size of your executable file. Larger executable files can take longer to load.

© Copyright IBM Corp. 1999, 2017

ARCH

You can use the highest value in the ARCH option to instruct the compiler to select from the largest set of instructions available under z/OS. This allows the compiler to generate the most optimal code.

REDUCE

The REDUCE option specifies that the compiler is permitted to reduce an assignment of a null string to a structure into a simple copy operation - even if that means padding bytes might be overwritten.

The REDUCE option will cause fewer lines of code to be generated for an assignment of a null string to a structure, and that will usually mean your compilation will be quicker and your code will run much faster. However, padding bytes might be zeroed out.

For instance, in the following structure, there is one byte of padding between `field11` and `field12`.

```
dcl
 1 sample ext,
   5  field10          bin fixed(31),
   5  field11          dec fixed(13),
   5  field12          bin fixed(31),
   5  field13          bin fixed(31),
   5  field14          bit(32),
   5  field15          bin fixed(31),
   5  field16          bit(32),
   5  field17          bin fixed(31);
```

Now consider the assignment `sample = '';`.

Under the NOREDUCE option, eight assignments will be generated, but the padding byte will be unchanged.

However, under REDUCE, the assignment will be reduced to three operations.

RULES

Most of the RULES suboptions affect only the severity with which certain coding practices, such as not declaring variables, are flagged and have no impact on performance. However, these suboptions do have an impact on performance.

IBM/ANS

When you use the RULES(IBM) option, the compiler supports scaled FIXED BINARY and, what is more important for performance, generates scaled FIXED BINARY results in some operations.

Under RULES(ANS), scaled FIXED BINARY is not supported and scaled FIXED BINARY results are never generated. This means that the code generated under RULES(ANS) always runs at least as fast as the code generated under RULES(IBM), and sometimes runs faster.

For example, consider the following code fragment:

```
dcl (i,j,k) fixed bin(15);
   .
   .
   .
i = j / k;
```

Under RULES(IBM), the result of the division has the attributes FIXED BIN(31,16). This means that a shift instruction is required before the division and several more instructions are needed to perform the assignment.

Under RULES(ANS), the result of the division has the attributes FIXED BIN(15,0). This means that a shift is not needed before the division, and no extra instructions are needed to perform the assignment.

(NO)LAXCTL

Under RULES(LAXCTL), a CONTROLLED variable can be declared with constant extents and yet allocated with different extents. However, this coding practice severely impacts performance. It is recommended that you use the RULES(NOLAXCTL) option to disallow such practice.

For instance, under RULES(LAXCTL), you can declare a structure as follows:

```
dcl
  1 a controlled,
    2 b char(17),
    2 c char(29);
```

However, you can then allocate it as follows:

```
allocate
  1 a,
    2 b char(170),
    2 c char(290);
```

Whenever the compiler sees a reference to the structure `A` or to any member of that structure, the compiler is forced to assume that it knows nothing about the lengths, dimensions, or offsets of the fields in the structure. This can severely degrade performance.

Under RULES(NOLAXCTL), if you want to allocate a CONTROLLED variable with a variable extent, that extent must be declared either with an asterisk or with a nonconstant expression. Consequently, under RULES(NOLAXCTL), when a CONTROLLED variable is declared with constant extents, the compiler can generate much better code for any reference to that variable.

PREFIX

The PREFIX option determines whether selected PL/I conditions are enabled by default. The default suboptions for PREFIX are set to conform to the PL/I language definition; however, overriding the defaults can have a significant effect on the performance of your program.

These are the default suboptions:

CONVERSION
INVALIDOP
FIXEDOVERFLOW
OVERFLOW
INVALIDOP
NOSIZE
NOSTRINGRANGE
NOSTRINGSIZE
NOSUBSCRIPTRANGE
UNDERFLOW
ZERODIVIDE

By specifying the SIZE, STRINGRANGE, STRINGSIZE, or SUBSCRIPTRANGE suboptions, the compiler generates extra code that helps you pinpoint various problem areas in your source that would otherwise be hard to find. This extra code, however, can slow program performance significantly.

CONVERSION

When you disable the CONVERSION condition, some character-to-numeric conversions are done inline and without checking the validity of the source; therefore, specifying NOCONVERSION also affects program performance.

FIXEDOVERFLOW

On some platforms, the FIXEDOVERFLOW condition is raised by the hardware and the compiler does not need to generate any extra code to detect it.

DEFAULT

Using the DEFAULT option, you can select attribute defaults. As is true with the PREFIX option, the suboptions for DEFAULT are set to conform to the PL/I language definition. Changing the defaults in some instances can affect performance.

Some of the suboptions, such as IBM/ANS and ASSIGNABLE/NONASSIGNABLE, have no effect on program performance. But other suboptions can affect performance to varying degrees and, if applied inappropriately, can make your program invalid. The following topics provide more information about some suboptions of more importance.

BYADDR or BYVALUE

When the DEFAULT(BYADDR) option is in effect, arguments are passed by reference (as required by PL/I) unless an attribute in an entry declaration indicates otherwise. As arguments are passed by reference, the address of the argument is passed from one routine (calling routine) to another (called routine) as the variable itself is passed. Any change made to the argument while in the called routine is reflected in the calling routine when it resumes execution.

Program logic often depends on passing variables by reference. Passing a variable by reference, however, can hinder performance in two ways:

1. Every reference to that parameter requires an extra instruction.
2. Because the address of the variable is passed to another routine, the compiler is forced to make assumptions about when that variable might change and generate very conservative code for any reference to that variable.

Consequently, you should pass parameters by value using the BYVALUE suboption whenever your program logic allows. Even if you use the BYADDR attribute to indicate that one parameter should be passed by reference, you can use the DEFAULT(BYVALUE) option to ensure that all other parameters are passed by value.

A BYVALUE argument should be one that could reasonably be passed in a register. Hence its type should be one of the following:

- REAL FIXED BIN
- REAL FLOAT
- POINTER

- OFFSET
- HANDLE
- LIMITED ENTRY
- FILE
- ORDINAL
- CHAR(1)
- WCHAR(1)
- ALIGNED BIT(*n*) with *n* less than or equal to 8

Moreover, when using BYVALUE parameters with assembler code, you must note that any BYVALUE parameter that does not require a multiple of 4 bytes will be widened so that 4 bytes are used in the parameter list.

If a procedure receives and modifies only one parameter that is passed by BYADDR, consider converting the procedure to a function that receives that parameter by value. The function would then end with a RETURN statement containing the updated value of the parameter.

Procedure with BYADDR parameter

```
a: proc( parm1, parm2, ..., parmN );

  dcl parm1 byaddr  ...;
  dcl parm2 byvalue ...;
   .
   .
   .
  dcl parmN byvalue ...;

  /* program logic */

end;
```

Faster, equivalent function with BYVALUE parameter

```
a: proc( parm1, parm2, ..., parmN )
   returns( ... /* attributes of parm1 */ );

  dcl parm1 byvalue  ...;
  dcl parm2 byvalue ...;
   .
   .
   .
  dcl parmN byvalue ...;

  /* program logic */

  return( parm1 );

end;
```

(NON)CONNECTED

The DEFAULT(NONCONNECTED) option indicates that the compiler assumes that any aggregate parameters are NONCONNECTED. References to elements of NONCONNECTED aggregate parameters require the compiler to generate code to access the parameter's descriptor, even if the aggregate is declared with constant extents.

The compiler does not generate these instructions if the aggregate parameter has constant extents and is CONNECTED. Consequently, if your application never passes nonconnected parameters, your code is more optimal if you use the DEFAULT(CONNECTED) option.

(NO)DESCRIPTOR

The DEFAULT(DESCRIPTOR) option indicates that, by default, a descriptor is passed for any string, area, or aggregate parameter; however, the descriptor is used only if the parameter has nonconstant extents or if the parameter is an array with the NONCONNECTED attribute.

In this case, the instructions and space required to pass the descriptor provide no benefit and incur substantial cost (the size of a structure descriptor is often greater than size of the structure itself). Consequently, by specifying DEFAULT(NODESCRIPTOR) and using OPTIONS(DESCRIPTOR) only as needed on PROCEDURE statements and ENTRY declarations, your code runs more optimally.

(NO)INLINE

The NOINLINE suboption indicates that procedures and begin blocks should not be inlined. Inlining occurs only when you specify optimization.

Inlining user code eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer, resulting in faster code performance. Inlining produces the best results when the overhead for the function is nontrivial, for example, when functions are called within nested loops. Inlining is also beneficial when the inlined function provides additional opportunities for optimization, such as when constant arguments are used.

For programs containing many procedures that are not nested, consider the following information:

- If the procedures are small and only called from a few places, you can increase performance by specifying INLINE.
- If the procedures are large and called from several places, inlining duplicates code throughout the program. This increase in the size of the program might offset any increase of speed. In this case, you might prefer to leave NOINLINE as the default and specify OPTIONS(INLINE) only on individually selected procedures.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends. Ensure that you have enough stack space for the local storage of the inlined functions.

LINKAGE

This LINKAGE suboption specifies the default linkage that the compiler should use when the LINKAGE suboption of the OPTIONS attribute or option for an entry has not been specified. The compiler supports various linkages, each with its unique performance characteristics.

When you invoke an ENTRY provided by an external entity (such as an operating system), you must use the linkage previously defined for that ENTRY.

As you create your own applications, however, you can choose the linkage convention. The OPTLINK linkage is strongly recommended because it provides significantly better performance than other linkage conventions.

(RE)ORDER

The DEFAULT(ORDER) option indicates that the ORDER option is applied to every block, meaning that variables in that block referenced in ON-units (or blocks dynamically descendant from ON-units) have their latest values. This effectively prohibits almost all optimization on such variables.

Consequently, if your program logic allows, use DEFAULT(REORDER) to generate superior code.

NOOVERLAP

The DEFAULT(NOOVERALP) option lets the compiler assume that the source and target in an assignment do not overlap, and it can therefore generate smaller and faster code.

However, if you use this option, you must ensure that the source and target in assignment do not overlap. For example, under the DEFAULT(NOOVERLAP) option, the assignment in this example is invalid:

```
dcl c char(20);
substr(c,2,5) = substr(c,1,5);
```

RETURNS(BYVALUE) or RETURNS(BYADDR)

When the DEFAULT(RETURNS(BYVALUE)) option is in effect, the BYVALUE attribute is applied to all RETURNS description lists that do not specify BYADDR. This means that these functions return values in registers, when possible, in order to produce the most optimal code.

Summary of compiler options that improve performance

In summary, the following options (if appropriate for your application) can improve performance:

- OPTIMIZE(3)
- ARCH(10)
- REDUCE
- RULES(ANS NOLAXCTL)
- DEFAULT with the following suboptions
  - BYVALUE
  - CONNECTED
  - NODESCRIPTOR
  - INLINE
  - LINKAGE(OPTLINK)
  - REORDER
  - NOOVERLAP
  - RETURNS(BYVALUE)

Coding for better performance

As you write code, there is generally more than one correct way to accomplish a given task. Many important factors influence the coding style you choose, including readability and maintainability. The following topics discuss choices that you can make while coding that potentially affect the performance of your program.

DATA-directed input and output

Using GET DATA and PUT DATA statements for debugging can prove very helpful. When you use these statements, however, you generally pay the price of decreased performance. This cost to performance is usually very high when you use either GET DATA or PUT DATA without a variable list.

Many programmers use PUT DATA statements in their ON ERROR code as illustrated in the following example:

```
on error
  begin;
    on error system;
   .
   .
   .
    put data;
   .
   .
   .
  end;
```

In this case, the program would perform more optimally by including a list of selected variables with the PUT DATA statement.

The ON ERROR block in the previous example contained an ON ERROR system statement before the PUT DATA statement. This prevents the program from getting caught in an infinite loop if an error occurs in the PUT DATA statement (which could occur if any variables to be listed contained invalid FIXED DECIMAL values) or elsewhere in the ON ERROR block.

Input-only parameters

If a procedure has a BYADDR parameter that it uses as input only, it is best to declare that parameter as NONASSIGNABLE (rather than letting it get the default attribute of ASSIGNABLE). If that procedure is later called with a constant for that parameter, the compiler can put that constant in static storage and pass the address of that static area.

This practice is particularly useful for strings and other parameters that cannot be passed in registers (input-only parameters that can be passed in registers are best declared as BYVALUE).

In the following declaration, for instance, the first parameter to `getenv` is an input-only CHAR VARYINGZ string:

```
dcl getenv     entry( char(*) varyingz nonasgn byaddr,
                      pointer byaddr )
               returns( native fixed bin(31) optional )
               options( nodescriptor );
```

If this function is invoked with the string `IBM_OPTIONS`, the compiler can pass the address of that string rather than assigning it to a compiler-generated temporary storage area and passing the address of that area.

GOTO statements

A GOTO statement that uses either a label in another block or a label variable severely limits optimizations that the compiler might perform.

If a label array is initialized and declared AUTOMATIC, either implicitly or explicitly, any GOTO to an element of that array will hinder optimization. However, if the array is declared as STATIC, the compiler assumes the CONSTANT attribute for it and no optimization is hindered.

String assignments

When one string is assigned to another, the compiler ensures that the target has the correct value even if the source and target overlap, and that the source string is truncated if it is longer than the target. This assurance comes at the cost of some extra instructions.

The compiler attempts to generate these extra instructions only when necessary, but often you, as the programmer, know they are not necessary when the compiler cannot be sure. For instance, if the source and target are based character strings and you know they cannot overlap, you could use the PLIMOVE built-in function to eliminate the extra code the compiler would otherwise be forced to generate.

In the following example, faster code is generated for the second assignment statement:

```
dcl based_Str   char(64) based( null() );
dcl target_Addr pointer;
dcl source_Addr pointer;

target_Addr->based_Str = source_Addr->based_Str;

call plimove( target_Addr, source_Addr, stg(based_Str) );
```

If you have any doubts about whether the source and target might overlap or whether the target is big enough to hold the source, you should not use the PLIMOVE built-in.

Loop control variables

Program performance improves if you define your loop control variables appropriately.

For better program performance, use one of the following types for your loop control variables. You should rarely, if ever, use other types of variables.

FIXED BINARY with zero scale factor
FLOAT
ORDINAL
HANDLE
POINTER
OFFSET

Performance also improves if loop control variables are not members of arrays, structures, or unions. The compiler issues a warning message when they are. Loop control variables that are AUTOMATIC and not used for any other purpose give you the optimal code generation.

If a loop control variable is a FIXED BINARY, performance is best if it has precision 31 and is SIGNED.

Performance is decreased if your program depends not only on the value of a loop control variable, but also on its address. For example, the performance is decreased if the ADDR built-in function is applied to the variable or if the variable is passed by reference (BYADDR) to another routine.

PACKAGEs versus nested PROCEDUREs

Calling nested procedures requires that an extra *hidden parameter* (the backchain pointer) is passed. As a result, the fewer nested procedures that your application contains, the faster it runs.

To improve the performance of your application, you can convert a mother-daughter pair of nested procedures into level-1 sister procedures inside of a package. This conversion is possible if your nested procedure does not rely on any of the automatic and internal static variables declared in its parent procedures.

If procedure b in "Example with nested procedures" does not use any of the variables declared in a, you can improve the performance of both procedures by reorganizing them into the package illustrated in "Example with packaged procedures."

Example with nested procedures

```
a: proc;

  dcl (i,j,k) fixed bin;
  dcl ib      based fixed bin;
   .
   .
   .
  call b( addr(i) );
   .
   .
   .
  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;
end;
```

Example with packaged procedures

```
p: package exports( a );

  dcl ib      based fixed bin;

  a: proc;

    dcl (i,j,k) fixed bin;
   .
   .
   .
    call b( addr(i) );
   .
   .
   .
  end;

  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;

end p;
```

REDUCIBLE functions

REDUCIBLE indicates that a procedure or an entry need not be invoked multiple times if the argument(s) stays unchanged, and that the invocation of the procedure has no side effects.

For example, a user-written function that computes a result based on unchanging data should be declared REDUCIBLE. A function that computes a result based on changing data, such as a random number or time of day, should be declared IRREDUCIBLE.

In the following example, `f` is invoked only once because REDUCIBLE is part of the declaration. If IRREDUCIBLE had been used in the declaration, `f` would be invoked twice.

```
dcl (f) entry options( reducible ) returns( fixed bin );

select;
  when( f(x) < 0 )
   .
   .
   .
  when( f(x) > 0 )
   .
   .
   .
  otherwise
   .
   .
   .
end;
```

DESCLOCATOR or DESCLIST

When the DEFAULT(DESCLOCATOR) option is in effect, the compiler passes arguments requiring descriptors (such as strings and structures) through a descriptor locator in much the same way that the old compiler did.

This option allows you to invoke an entry point that is not always passed all of the arguments that it declares.

This option also allows you to continue the somewhat unwise programming practice of passing a structure and receiving it as a pointer.

However, the code generated by the compiler for DEFAULT(DESCLOCATOR) might, in some situations, perform less well than that for DEFAULT(DESCLIST).

Related information:

Chapter 25, “PL/I descriptors,” on page 497
This chapter describes PL/I parameter passing conventions between PL/I routines at run time.

DEFINED versus UNION

The UNION attribute is more powerful than the DEFINED attribute and provides more functions. In addition, the compiler generates better code for union references.

In the following example, the pair of variables `b3` and `b4` perform the same function as `b1` and `b2`, but the compiler generates more optimal code for the pair in the union.

```
dcl b1 bit(32);
dcl b2 bit(16) def b1;

dcl
  1 * union,
    2 b3 bit(32),
    2 b4 bit(16);
```

Code that uses UNIONs instead of the DEFINED attribute is subject to less misinterpretation. Variable declarations in unions are in a single location making it easy to realize that when one member of the union changes, all of the others change also. This dynamic change is less obvious in declarations that use DEFINED variables because the declare statements can be several lines apart.

Named constants versus static variables

You can define named constants by declaring a variable with the VALUE attribute. If you use static variables with the INITIAL attribute and you do not alter the variable, you should declare the variable a named constant by using the VALUE attribute. The compiler does not treat NONASSIGNABLE scalar STATIC variables as true named constants.

The compiler generates better code whenever expressions are evaluated during compilation, so you can use named constants to produce efficient code with no loss in readability. For example, identical object code is produced for the two usages of the VERIFY built-in function in the following example:

```
dcl numeric char  value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );
```

The following examples illustrate how you can use the VALUE attribute to get optimal code without sacrificing readability.

Example with optimal code but no meaningful names

```
dcl  x  bit(8) aligned;

select( x );
  when( '01'b4 )
    .
    .
    .
  when( '02'b4 )
    .
    .
    .
  when( '03'b4 )
    .
    .
    .
end;
```

Example with meaningful names but not optimal code

```
dcl (  a1  init( '01'b4)
      ,a2  init( '02'b4)
      ,a3  init( '03'b4)
      ,a4  init( '04'b4)
      ,a5  init( '05'b4)
    ) bit(8) aligned static nonassignable;

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
    .
    .
    .
  when( a2 )
    .
```

```
   .
   .
  when( a3 )
   .
   .
   .
end;
```

Example with optimal code and meaningful names

```
dcl (  a1  value( '01'b4)
      ,a2  value( '02'b4)
      ,a3  value( '03'b4)
      ,a4  value( '04'b4)
      ,a5  value( '05'b4)
    ) bit(8);

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
   .
   .
   .
  when( a2 )
   .
   .
   .
  when( a3 )
   .
   .
   .
end;
```

Avoiding calls to library routines

The bitwise operations (prefix NOT, infix AND, infix OR, and infix EXCLUSIVE OR) are often evaluated by calls to library routines.

These operations are, however, handled without a library call if either of the following conditions is true:

- Both operands are bit(1).
- Both operands are aligned and have the same constant length.

For certain assignments, expressions, and built-in function references, the compiler generates calls to library routines. If you avoid these calls, your code generally runs faster.

To help you determine when the compiler generates such calls, the compiler generates a message whenever a conversion is done by a library routine.

When your code refers to a member of a BASED structure with REFER, the compiler often has to generate one or more calls to a library routine to map the structure at run time. These calls can be expensive, and so when the compiler makes these calls, it will issue a message so that you can locate these potential hot-spots in your code.

If you do have code that uses BASED structures with REFER, which the compiler flags with this message, you might get better performance by passing the structure to a subroutine that declares a corresponding structure with * extents. This will cause the structure to be mapped once at the CALL statement, but there will no further remappings when it is accessed in the called subroutine.

Preloading library routines

The PL/I library contains one RMODE 24 routine, IBMPOIOA, that is used for low-level system i/o functions. If your code does RECORD i/o or uses SYSPRINT as a STREAM OUTPUT file (without compiling with the STDSYS option), you will significantly improve your performance if you preload this routine or put it into the (E)LPA.

Part 4. Using interfaces to other products

© Copyright IBM Corp. 1999, 2017

Chapter 16. Using the Sort program

The compiler provides an interface called PLISRT*x* (*x* = A, B, C, or D) that allows you to make use of the IBM-supplied Sort programs.

To use the Sort program with PLISRT*x*, you must do the following:

1. Include a call to one of the entry points of PLISRT*x*, passing it the information on the fields to be sorted. This information includes the length of the records, the maximum amount of storage to use, the name of a variable to be used as a return code, and other information required to carry out the sort.
2. Specify the data sets required by the Sort program in JCL DD statements.

When used from PL/I, the Sort program sorts records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted can be either a data set or a user-written PL/I procedure that the Sort program will call each time a record is required for the sort. Similarly, the destination of the sort can be a data set or a PL/I procedure that handles the sorted records.

Using PL/I procedures allows processing to be done before or after the sort itself, thus allowing a complete sorting operation to be handled completely by a call to the sort interface. It is important to understand that the PL/I procedures handling input or output are called from the Sort program itself and will effectively become part of it.

PL/I can operate with DFSORT or a program with the same interface. DFSORT is a release of the program product 5740-SM1. DFSORT has many built-in features you can use to eliminate the need for writing program logic (for example, INCLUDE, OMIT, OUTREC, and SUM statement plus the many ICETOOL operators). See *DFSORT Application Programming Guide* for details and *Getting Started with DFSORT* for a tutorial.

The following information applies to DFSORT. Because you can use programs other than DFSORT, the actual capabilities and restrictions vary. For these capabilities and restrictions, see *DFSORT Application Programming Guide* or the equivalent publication for your sort product.

To use the Sort program, you must include the correct PL/I statements in your source program and specify the correct data sets in your JCL.

Preparing to use Sort

Before using Sort, you must determine the type of sort you require, the length and format of the sorting fields in the data, the length of your data records, and the amount of auxiliary and main storage you will allow for sorting.

To determine the PLISRT*x* entry point that you will use, you must decide the source of your unsorted data, and the destination of your sorted data. You must choose between data sets and PL/I subroutines. Using data sets is simpler to understand and gives faster performance. Using PL/I subroutines gives you more flexibility and more function, enabling you to manipulate or print the data before it

© Copyright IBM Corp. 1999, 2017

is sorted, and to make immediate use of it in its sorted form. If you decide to use an input or output handling subroutine, see "Data input and output handling routines" on page 356.

The entry points and the source and destination of data are as follows:

Entry point	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine
PLISRTD	Subroutine	Subroutine

Having determined the entry point you are using, you must now determine these about your data set:

- The position of the sorting fields; these can be either the complete record or any part or parts of it
- The type of data these fields represent, for example, character or binary
- Whether you want the sort on each field to be in ascending or descending order
- Whether you want equal records to be retained in the order of the input, or whether their order can be altered during sorting

Specify these options on the SORT statement, which is the first argument to PLISRT*x*. Then, you must determine these about the records to be sorted:

- Whether the record format is fixed or varying
- The length of the record, which is the maximum length for varying

Specify these on the RECORD statement, which is the second argument to PLISRT*x*.

Finally, you must decide on the amount of main and auxiliary storage you will allow for the Sort program. For further details, see "Determining storage needed for Sort" on page 351.

Choosing the type of Sort

In your PL/I program, you specify a sort by using a CALL statement to the sort interface subroutine PLISRT*x*. This subroutine has four entry points: *x*=A, B, C, and D. Each specifies a different source for the unsorted data and destination for the data when it has been sorted.

For example, a call to PLISRTA specifies that the unsorted data (the input to sort) is on a data set, and that the sorted data (the output from sort) is to be placed on another data set. The CALL PLISRT*x* statement must contain an argument list giving the Sort program information about the data set to be sorted, the fields on which it is to be sorted, the amount of space available, the name of a variable into which Sort will place a return code indicating the success or failure of the sort, and the name of any output or input handling procedure that can be used.

The sort interface routine builds an argument list for the Sort program from the information supplied by the PLISRT*x* argument list and the choice of PLISRT*x* entry point. Control is then transferred to the Sort program. If you have specified an output- or input-handling routine, this will be called by the Sort program as many times as is necessary to handle each of the unsorted or sorted records. When the sort operation is complete, the Sort program returns to the PL/I calling

procedure communicating its success or failure in a return code, which is placed in one of the arguments passed to the interface routine. The return code can then be tested in the PL/I routine to discover whether processing should continue. Figure 52 on page 348 is a simplified flowchart showing this operation.

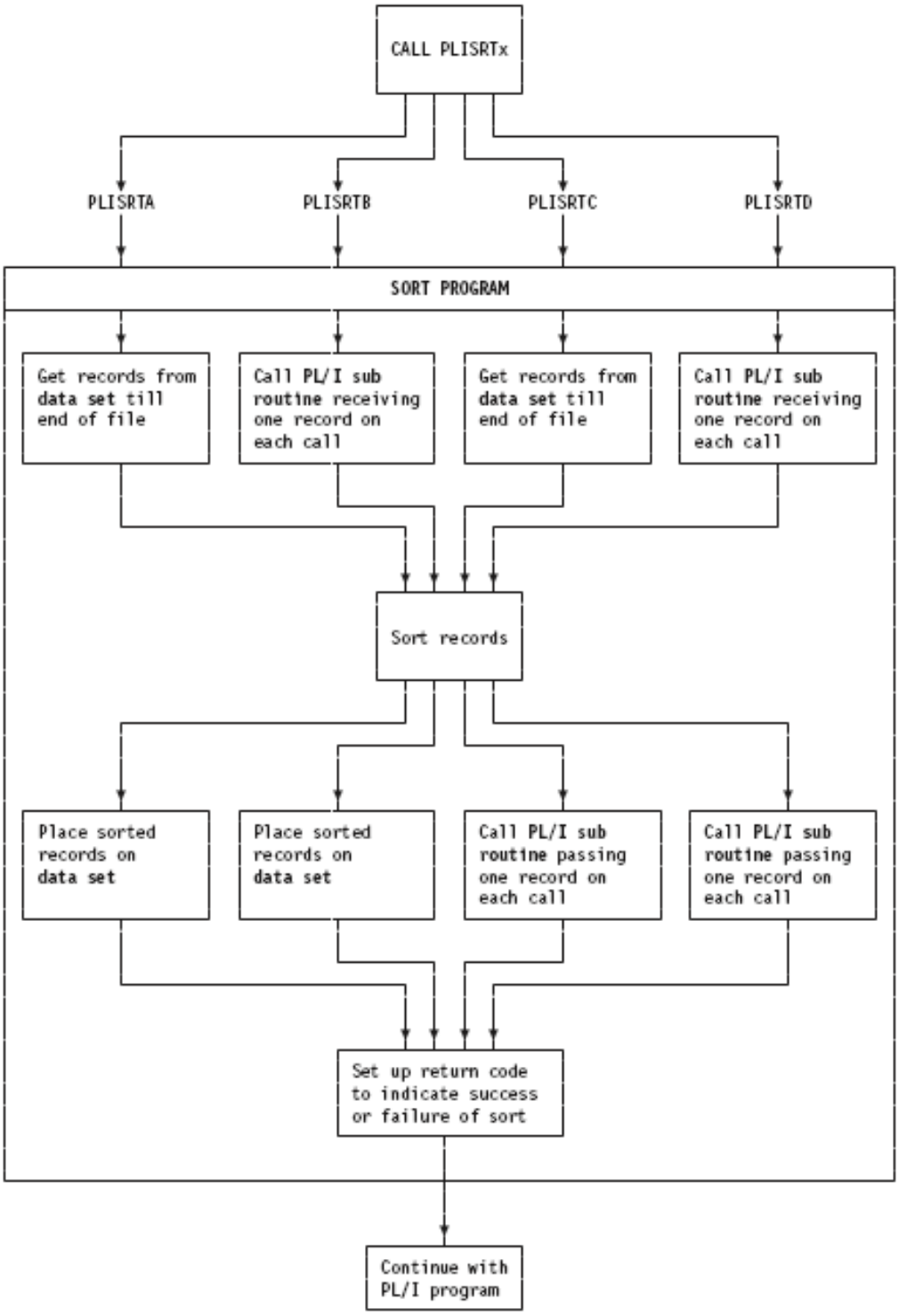


Figure 52. Flow of control for Sort program

Within the Sort program itself, the flow of control between the Sort program and input- and output-handling routines is controlled by return codes. The Sort

program calls these routines at the appropriate point in its processing. (Within the Sort program, and its associated documentation, these routines are known as *user exits*. The routine that passes input to be sorted is the E15 sort user exit. The routine that processes sorted output is the E35 sort user exit.) From the routines, Sort expects a return code indicating either that it should call the routine again, or that it should continue with the next stage of processing.

You must remember the following important points about Sort:

- It is a self-contained program that handles the complete sort operation.
- It communicates with the caller, and with the user exits that it calls, through return codes.

Specifying the sorting field

This topic describes the required PL/I statements that you must specify to use Sort from PL/I. The SORT statement is the first argument to PLISRT*x*.

The syntax of the SORT statement must be a character string expression that takes the form:

```
'bSORTbFIELDS=(start1,length1,form1,seq1,
...startn,lengthn,formn,seqn)[,other options]b'
```

See the following example:

```
' SORT FIELDS=(1,10,CH,A) '
```

b Represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

start,length,form,seq
Defines a sorting field. You can specify any number of sorting fields, but there is a limit on the total length of the fields. If more than one field is to be sorted on, the records are sorted first according to the first field, and then those that are of equal value are sorted according to the second field, and so on. If all the sorting values are equal, the order of equal records will be arbitrary unless you use the EQUALS option. (See later in this definition list.) Fields can overlay each other.

For DFSORT (5740-SM1), the maximum total length of the sorting fields is restricted to 4092 bytes and all sorting fields must be within 4092 bytes of the start of the record. Other sort products might have different restrictions.

start Specifies the starting position within the record. Specify the value in bytes except for binary data where you can use a "byte.bit" notation. The first byte in a string is considered to be byte 1, and the first bit is bit 0. (Thus the second bit in byte 2 is referred to as 2.1.) For varying length records, you must include the 4-byte length prefix, making 5 the first byte of data.

length Specifies the length of the sorting field. Specify the value in bytes except for binary where you can use "byte.bit" notation. The length of sorting fields is restricted according to their data type.

form Specifies the format of the data. This is the format assumed for the purpose of sorting. All data passed between PL/I routines and Sort must be in the form of character strings. The main data types and the restrictions on their length are shown below. Additional data types are available for special-purpose sorts. See the *DFSORT Application Programming Guide* or the equivalent publication for your sort product.

Code	Data type and length
CH	Character 1–4096
ZD	Zoned decimal, signed 1–32
PD	Packed decimal, signed 1–32
FI	Fixed point, signed 1–256
BI	Binary, unsigned 1 bit to 4092 bytes
FL	Floating-point, signed 1–256

The sum of the lengths of all fields must not exceed 4092 bytes.

seq Specifies the sequence in which the data will be sorted:

A Ascending (that is, 1,2,3,...)

D Descending (that is, ...,3,2,1)

Note: You cannot specify E, because PL/I does not provide a method of passing a user-supplied sequence.

other options

You can specify a number of other options, depending on your Sort program. You must separate them from the FIELDS operand and from each other by commas. Do not place blanks between operands.

FILSZ=y

Specifies the number of records in the sort and allows for optimization by Sort. If `y` is only approximate, E should precede `y`.

SKIPREC=y

Specifies that `y` records at the start of the input file are to be ignored before Sort starts sorting the remaining records.

CKPT or CHKPT

Specifies that checkpoints are to be taken. If you use this option, you must provide a SORTCKPT data set. In addition, when you install DFSORT, you must specify the 16NCKPT=NO installation option.

EQUALS|NOEQUALS

Specifies whether the order of equal records will be preserved as it was in the input (EQUALS) or will be arbitrary (NOEQUALS). You could improve sort performance by using the NOEQUALS. The default option is chosen when Sort is installed. The IBM-supplied default is NOEQUALS.

DYNALLOC=(d,n)

(OS/VS Sort only) Specifies that the program dynamically allocates intermediate storage.

d Specifies the device type (3380, and so on).

n Specifies the number of work areas.

Specifying the records to be sorted

This topic describes the required PL/I statements that you must specify to use Sort from PL/I. The RECORD statement is the second argument to PLISRT*x*.

The syntax of the RECORD statement must be a character string expression, which, when evaluated, takes the following syntax:

```
'bRECORDbTYPE=rectype[,LENGTH=(L1,[,,L4,L5])]b'
```

See the following example:

```
' RECORD TYPE=F,LENGTH=(80) '
```

b Represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

TYPE

Specifies the type of record as follows:

F Fixed length

V Varying length EBCDIC

D Varying length ASCII

Even when you use input and output routines to handle the unsorted and sorted data, you must specify the record type as it applies to the work data sets used by Sort.

If varying length strings are passed to Sort from an input routine (E15 exit), you should normally specify V as a record format. However, if you specify F, the records are padded to the maximum length with blanks.

LENGTH

Specifies the length of the record to be sorted. You can omit LENGTH if you use PLISRTA or PLISRTC, because the length will be taken from the input data set. Note that there is a restriction on the maximum and minimum length of the record that can be sorted. For varying length records, you must include the 4-byte prefix.

L1 Specifies the length of the record to be sorted. For VSAM data sets sorted as varying records, it is the maximum record size + 4.

„ Represents two arguments that are not applicable to Sort when called from PL/I. You must include the commas if the arguments that follow are used.

L4 Specifies the minimum length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

L5 Specifies the modal (most common) length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

Maximum record lengths: The length of a record can never exceed the maximum length specified by the user. The maximum record length for variable length records is 32756 bytes, and for fixed length records, it is 32760 bytes.

Determining storage needed for Sort

Sort requires both main and auxiliary storage.

Main storage

The minimum main storage for DFSORT is 88K bytes, but for best performance, more storage (on the order of 1 megabyte or more) is recommended. DFSORT can take advantage of storage above 16M virtual or extended architecture processors. Under z/OS, DFSORT can also take advantage of expanded storage. You can specify that Sort use the maximum amount of storage available by passing a storage parameter in the following manner:

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B||UNSPEC('MAX');
CALL PLISRTA
    (' SORT FIELDS=(1,80,CH,A) ',
```

```
' RECORD TYPE=F,LENGTH=(80) ',
  MAXSTOR,
  RETCODE,
  'TASK');
```

If files are opened in E15 or E35 exit routines, enough residual storage should be allowed for the files to open successfully.

Auxiliary storage

Calculating the minimum auxiliary storage for a particular sorting operation is a complicated task. To achieve maximum efficiency with auxiliary storage, use direct access storage devices (DASDs) whenever possible. For more information about improving program efficiency, see the *DFSORT Application Programming Guide*, particularly the information about dynamic allocation of workspace that allows DFSORT to determine the auxiliary storage needed and allocate it for you.

If you are interested only in providing enough storage to ensure that the sort will work, make the total size of the SORTWK data sets large enough to hold three sets of the records being sorted. (You will not gain any advantage by specifying more than three if you have enough space in three data sets.)

However, because this suggestion is an approximation, it might not work, so you should check the sort manuals. If this suggestion does work, you will probably have wasted space.

Calling the Sort program

You should write the CALL PLISRT*x* statement with some care. This topic lists the entry points and arguments that you can use.

Table 34. The entry points and arguments to PLISRTx (x = A, B, C, or D)

Entry points	Arguments
PLISRTA Sort input: data set Sort output: data set	(sort statement,record statement,storage,return code [,data set prefix,message level, sort technique])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(sort statement,record statement,storage,return code,input routine [,data set prefix,message level,sort technique])
PLISRTC Sort input: data set Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,output routine [,data set prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,input routine,output routine[,data set prefix,message level,sort technique])
Sort statement	Character string expression containing the Sort program SORT statement. Describes sorting fields and format. See “Specifying the sorting field” on page 349.
Record statement	Character string expression containing the Sort program RECORD statement. Describes the length and record format of data. See “Specifying the records to be sorted” on page 350.
Storage	Fixed binary expression giving maximum amount of main storage to be used by the Sort program. Must be >88K bytes for DFSORT. See also “Determining storage needed for Sort” on page 351.

Table 34. The entry points and arguments to PLISRTx (x = A, B, C, or D) (continued)

Entry points	Arguments
Return code	Fixed binary variable of precision (31,0) in which Sort places a return code when it has completed. The meaning of the return code is: 0=Sort successful 16=Sort failed 20=Sort message data set missing
Input routine	(PLISRTB and PLISRTD only.) Name of the PL/I external or internal procedure used to supply the records for the Sort program at sort exit E15.
Output routine	(PLISRTC and PLISRTD only.) Name of the PL/I external or internal procedure to which Sort passes the sorted records at sort exit E35.
Data set prefix	Character string expression of four characters that replaces the default prefix of 'SORT' in the names of the sort data sets SORTIN, SORTOUT, SORTWKnn and SORTCNTL, if used. Thus if the argument is “TASK”, the data sets TASKIN, TASKOUT, TASKWKnn, and TASKCNTL can be used. This facility enables multiple invocations of Sort to be made in the same job step. The four characters must start with an alphabetic character and must not be one of the reserved names PEER, BALN, CRCX, OSCL, POLY, DIAG, SYSC, or LIST. You must code a null string for this argument if you require either of the following arguments but do not require this argument.
Message level	Character string expression of two characters indicating how Sort's diagnostic messages are to be handled, as follows: NO No messages to SYSOUT AP All messages to SYSOUT CP Critical messages to SYSOUT SYSOUT will normally be allocated to the printer, hence the use of the mnemonic letter “P”. Other codes are also allowed for certain of the Sort programs. For further details on these codes, see *DFSORT Application Programming Guide*. You must code a null string for this argument if you require the following argument but you do not require this argument.
Sort technique	(This is not used by DFSORT; it appears for compatibility reasons only.) Character string of length 4 that indicates the type of sort to be carried out, as follows: PEER Peerage sort BALN Balanced CRCX Criss-cross sort OSCL Oscillating POLY Polyphase sort Normally the Sort program will analyze the amount of space available and choose the most effective technique without any action from you. You should use this argument only as a bypass for sorting problems or when you are certain that performance could be improved by another technique. See *DFSORT Application Programming Guide* for further information.

The following examples indicate the form that the CALL PLISRT*x* statement normally takes.

Example 1

This example shows a call to PLISRTA, sorting 80-byte records from SORTIN to SORTOUT, using 1048576 (1 megabyte) of storage and a return code, RETCODE, declared as FIXED BINARY (31,0).

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE);
```

Example 2

This example is the same as example 1 except that the input, output, and work data sets are called TASKIN, TASKOUT, TASKWK01, and so forth.

This might occur if Sort was being called twice in one job step.

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE,
                'TASK');
```

Example 3

This example is the same as example 1 except that the sort is to be undertaken on two fields (first, bytes 1 to 10, which are characters, and then, if these are equal, bytes 11 and 12, which contain a binary field).

Both fields are to be sorted in ascending order.

```
CALL PLISRTA (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE);
```

Example 4

This example shows a call to PLISRTB.

The input is to be passed to Sort by the PL/I routine PUTIN. The sort is to be carried out on characters 1 to 10 of an 80-byte fixed length record.

```
CALL PLISRTB (' SORT FIELDS=(1,10,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE,
                PUTIN);
```

Example 5

This example shows a call to PLISRTD.

The input is to be supplied by the PL/I routine PUTIN and the output is to be passed to the PL/I routine PUTOUT. The record to be sorted is 84 bytes varying (including the length prefix). It is to be sorted on bytes 1 through 5 of the data in ascending order, then if these fields are equal, on bytes 6 through 10 in descending order. (Note that the 4-byte length prefix is included so that the actual values used are 5 and 10 for the starting points.) If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

```
CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(84) ',
                1048576,
                RETCODE,
                PUTIN,       /*input routine (sort exit E15)*/
                PUTOUT);     /*output routine (sort exit E35)*/
```

Determining whether the Sort was successful

When the sort is completed, Sort sets a return code in the variable named in the fourth argument of the call to PLISRT*x*.

It then returns control to the statement that follows the CALL PLISRT*x* statement. The value returned indicates the success or failure of the sort as follows:

0 Sort successful
16 Sort failed
20 Sort message data set missing

You must declare the variable to which the return code is passed as FIXED BINARY (31,0). It is standard practice to test the value of the return code after the CALL PLISRT*x* statement and take appropriate action according to the success or failure of the operation.

See the following example (assuming that the return code is called RETCODE):

```
IF RETCODE¬=0 THEN DO;
  PUT DATA(RETCODE);
  SIGNAL ERROR;
END;
```

If the job step that follows the sort depends on the success or failure of the sort, you should set the value returned in the Sort program as the return code from the PL/I program. This return code is then available for the following job step. The PL/I return code is set by a call to PLIRETC. You can call PLIRETC with the value returned from Sort:

```
CALL PLIRETC(RETCODE);
```

You should not confuse this call to PLIRETC with the calls made in the input and output routines, where a return code is used for passing control information to Sort.

Establishing data sets for Sort

If DFSORT is installed in a library not known to the system, you must specify the DFSORT library in a JOBLIB or STEPLIB DD statement.

When you call Sort, the following sort data sets must not be open:

SYSOUT
A data set (normally the printer) on which messages from the Sort program will be written.

Sort work data sets: SORTWK01–SORTWK32

Note: If you specify more than 32 sort work data sets, DFSORT will only use the first 32.

******WK01–****WK32**
From 1 to 32 working data sets that are used in the sorting process

These must be direct access. For a discussion of space required and number of data sets, see "Determining storage needed for Sort" on page 351.

**** represents the four characters that you can specify as the data set prefix argument in calls to PLISRT*x*. This allows you to use data sets

with prefixes other than SORT. They must start with an alphabetic character and must not be the names PEER, BALN, CRCX, OSCL, POLY, SYSC, LIST, or DIAG.

Input data set: SORTIN

******IN**
The input data set that is used when PLISRTA and PLISRTC are called

See ****WK01–****WK32 for a detailed description.

Output data set: SORTOUT

******OUT**
The output data set that is used when PLISRTA and PLISRTB are called

See ****WK01–****WK32 for a detailed description.

Checkpoint data set: SORTCKPT
Data set that is used to hold checkpoint data, if the CKPT or CHKPT option is used in the SORT statement argument and the DFSORT 16NCKPT=NO installation option is specified

For information about this program DD statement, see *DFSORT Application Programming Guide*.

DFSPARM SORTCNTL
Data set from which additional or changed control statements can be read (optional)

For additional information about this program DD statement, see *DFSORT Application Programming Guide*.

See ****WK01–****WK32 for a detailed description.

Sort data input and output

The source of the data to be sorted is provided either directly from a data set or indirectly by a routine (Sort Exit E15) written by the user. Similarly, the destination of the sorted output is either a data set or a routine (Sort Exit E35) provided by the user.

PLISRTA is the simplest of all of the interfaces because it sorts from data set to data set. For an example of a PLISRTA program, see Figure 56 on page 361. Other interfaces require the input handling routine, the output handling routine, or both.

Data input and output handling routines

The input handling and output handling routines are called by Sort when PLISRTB, PLISRTC, or PLISRTD is used.

The routines must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRT*x*, they behave in the same way as ordinary internal procedures in respect of scope of names. The input and output procedure names must themselves be known in the procedure that makes the call to PLISRT*x*.

The routines are called individually for each record required by Sort or passed from Sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures will not retain their

values between calls. Consequently, items such as counters, which need to be retained from one call to the next, should either be declared as STATIC or be declared in the containing block.

The E15 and E35 sort exits must not be MAIN procedures.

E15—Input handling routine (Sort Exit E15)

Input routines are normally used to process the data in some way before it is sorted.

You can use input routines to print the data, as shown in Figure 57 on page 362 and Figure 59 on page 364, or to generate or manipulate the sorting fields to achieve the correct results.

The input handling routine is used by Sort when a call is made to either PLISRTB or PLISRTD. When Sort requires a record, it calls the input routine, which should return a record in character string format, with return code 12. This return code means that the record passed is to be included in the sort. Sort continues to call the routine until return code 8 is passed. Return code 8 means that all records have already been passed, and that Sort is not to call the routine again. If a record is returned when the return code is 8, it is ignored by Sort.

The data returned by the routine must be a character string. It can be fixed or varying. If it is varying, you should normally specify V as the record format in the RECORD statement, which is the second argument in the call to PLISRT*x*. However, you can specify F, in which case the string will be padded to its maximum length with blanks. The record is returned with a RETURN statement, and you must specify the RETURNS attribute in the PROCEDURE statement. The return code is set in a call to PLIRETC. A flowchart for a typical input routine is shown in "E15—Input handling routine (Sort Exit E15)."

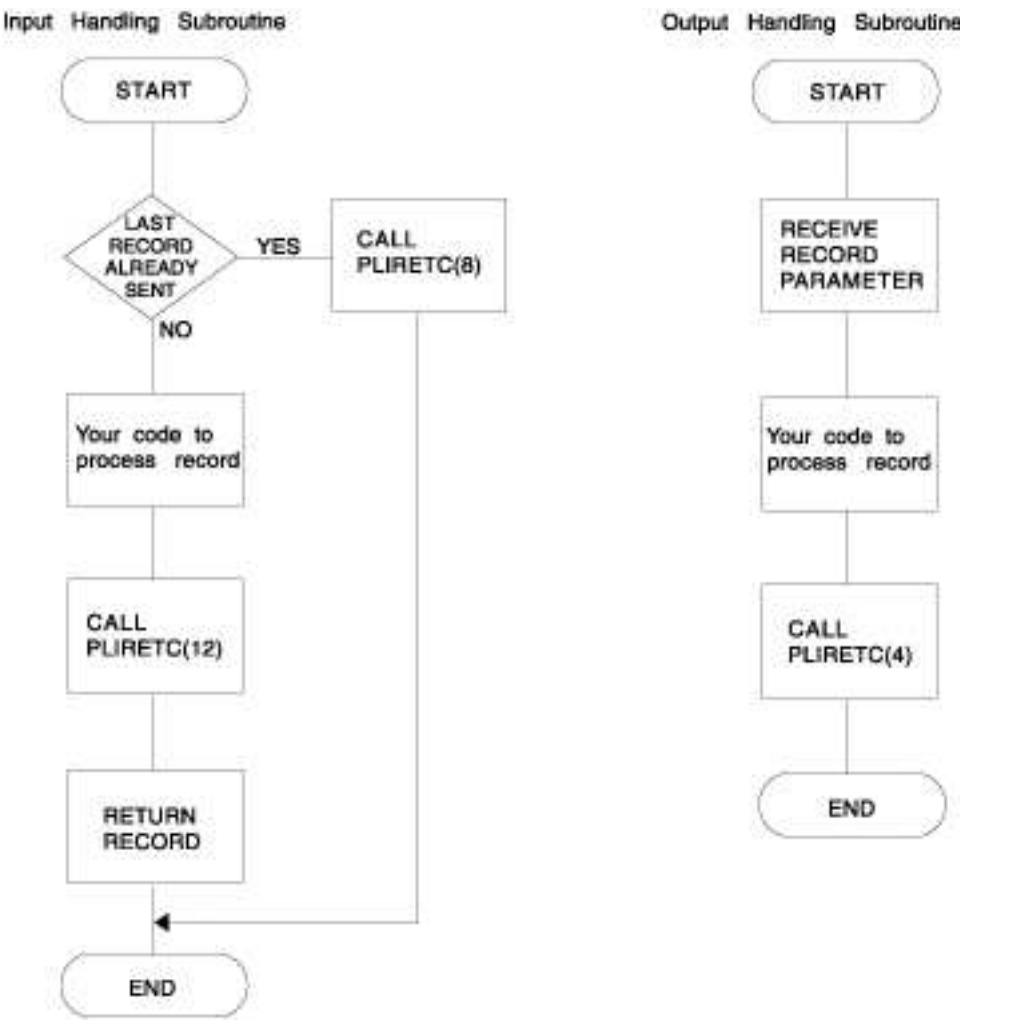


Figure 53. Flowcharts for input and output handling subroutines

Skeletal code for a typical input routine is shown in Figure 54 on page 359.

```
E15: PROC RETURNS (CHAR(80));
     /*-----------------------------------------------------------------*/
     /*RETURNS attribute must be used specifying length of data to be */
     /* sorted, maximum length if varying strings are passed to Sort. */
     /*-----------------------------------------------------------------*/
   DCL STRING CHAR(80); /*-------------------------------------------*/
                        /*A character string variable will normally be*/
                        /* required to return the data to Sort       */
                        /*-------------------------------------------*/

   IF LAST_RECORD_SENT THEN
     DO;
      /*-----------------------------------------------------------------*/
      /*A test must be made to see if all the records have been sent,  */
      /*if they have, a return code of 8 is set up and control returned*/
      /*to Sort                                                          */
      /*-----------------------------------------------------------------*/

        CALL PLIRETC(8);  /*-----------------------------------------*/
                          /* Set return code of 8, meaning last record */
                          /* already sent.                           */
                          /*-----------------------------------------*/
        RETURN('');
     END;

   ELSE
     DO;
      /*-------------------------------------------------*/
      /* If another record is to be sent to Sort, do the*/
      /* necessary processing, set a return code of 12  */
      /* by calling PLIRETC, and return the data as a   */
      /* character string to Sort                       */
      /*-------------------------------------------------*/

   ****(The code to do your processing goes here)

       CALL PLIRETC (12);  /*--------------------------------------*/
                           /* Set return code of 12, meaning this  */
                           /* record is to be included in the sort */
                           /*--------------------------------------*/
       RETURN (STRING);  /*Return data with RETURN statement*/
     END;

  END;   /*End of the input procedure*/
```

Figure 54. Skeletal code for an input procedure

For examples of an input routine, see Figure 57 on page 362 and Figure 59 on page 364.

In addition to return codes 12 (include current record in sort) and 8 (all records sent), Sort allows the use of return code 16 (Sort failed). This return code ends the sort and causes Sort to return to your PL/I program with return code 16.

Note: A call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When an output handling routine has been used, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 58 on page 363.

E35—Output handling routine (Sort Exit E35)

Output handling routines are normally used for any processing that is necessary after the sort.

For example, you can use output handling routines to print the sorted data, as shown in Figure 58 on page 363 and Figure 59 on page 364, or to use the sorted data to generate further information. The output handling routine is used by Sort when a call is made to PLISRTC or PLISRTD. When the records have been sorted, Sort passes them, one at a time, to the output handling routine. The output routine then processes them as required. When all the records have been passed, Sort sets up its return code and returns to the statement after the CALL PLISRT*x* statement. Sort does not indicate to the output handling routine that the last record has been reached. Any end-of-data handling must therefore be done in the procedure that calls PLISRT*x*.

The record is passed from Sort to the output routine as a character string, and you must declare a character string parameter in the output handling subroutine to receive the data. The output handling subroutine must also pass return code 4 to Sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

To stop the sort, pass return code 16 to Sort. This will result in Sort returning to the calling program with return code 16–Sort failed.

The record passed to the routine by Sort is a character string parameter. If you specify the record type as F in the second argument in the call to PLISRT*x*, you should declare the parameter with the length of the record. If you specify the record type as V, you should declare the parameter as adjustable, as in the following example:

```
DCL STRING CHAR(*);
```

Figure 60 on page 365 shows a program that sorts varying length records.

A flowchart for a typical output handling routine is given in "E15—Input handling routine (Sort Exit E15)" on page 357. Skeletal code for a typical output handling routine is shown in Figure 55.

```
E35: PROC(STRING);      /*The procedure must have a character string
                          parameter to receive the record from Sort*/

  DCL STRING CHAR(80);  /*Declaration of parameter*/

 (Your code goes here)

  CALL PLIRETC(4);   /*Pass return code to Sort indicating that the next
                       sorted record is to be passed to this procedure.*/
   END E35;          /*End of procedure returns control to Sort*/
```

Figure 55. Skeletal code for an output handling procedure

You should note that a call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When you have used an output handling routine, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the

argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples that follow this topic.

Calling PLISRTA example

This topic shows an example of a PLISRTA program.

After each time that the PL/I input- and output-handling routines communicate the return code information to the Sort program, the return code field is reset to zero; therefore, it is not used as a regular return code other than its specific use for the Sort program.

For details on handling conditions, especially those that occur during the input- and output-handling routines, see *z/OS Language Environment Programming Guide*.

```
//OPT14#7  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX106: PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                     1048576
                     RETURN_CODE);
     SELECT (RETURN_CODE);
       WHEN(0) PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT (
            'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
       CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/
       END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*
```

Figure 56. PLISRTA—sorting from input data set to output data set

Calling PLISRTB example

This topic shows an example of a PLISRTB program, which calls an input routine to get data and places sorted records on a data set.

```
//OPT14#8  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX107:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X);
     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER   PUT SKIP EDIT
           ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
     END /* select */;
     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E15X:   /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT
            STREAM AND PUTS THEM BEFORE THEY ARE SORTED*/
     PROC RETURNS (CHAR(80));
          DCL SYSIN FILE RECORD INPUT,
              INFIELD CHAR(80);

          ON ENDFILE(SYSIN) BEGIN;
             PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
             CALL PLIRETC(8);  /* signal that last record has
                                  already been sent to sort*/
             INFIELD = '';
             GOTO ENDE15;
             END;

           READ FILE (SYSIN) INTO (INFIELD);
           PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
           CALL PLIRETC(12);  /* request sort to include current
                                 record and return for more*/
    ENDE15:
          RETURN(INFIELD);
          END E15X;
  END EX107;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT  DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 57. PLISRTB—sorting from input handling routine to output data set

Calling PLISRTC example

This topic shows an example of a PLISRTC program, which sorts records in an input data set and calls an output handling routine to print sorted data.

```
//OPT14#9  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX108:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E35X);
        SELECT(RETURN_CODE);
          WHEN(0)  PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
          WHEN(16) PUT SKIP EDIT
              ('SORT FAILED, RETURN_CODE 16') (A);
          WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
          OTHER    PUT SKIP EDIT
              ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
         END /* select */;
     CALL PLIRETC (RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E35X:   /* output handling routine prints sorted records*/
       PROC (INREC);
          DCL INREC CHAR(80);
          PUT SKIP EDIT (INREC) (A);
          CALL PLIRETC(4); /*request next record from sort*/
       END E35X;
    END EX108;
/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT   DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 58. PLISRTC—sorting from input data set to output handling routine

Calling PLISRTD example

This topic shows an example of a PLISRTD program, which calls the input handling routine to print unsorted data and calls the output handling routine to print sorted data.

```
//OPT14#10  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX109:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);
     CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E15X,
                  E35X);

     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT
           ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;

     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

  E15X:   /* Input handling routine prints input before sorting*/
        PROC RETURNS(CHAR(80));
             DCL INFIELD CHAR(80);

             ON ENDFILE(SYSIN) BEGIN;
                PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT.  ',
                    'SORTED OUTPUT SHOULD FOLLOW')(A);
                CALL PLIRETC(8);  /* Signal end of input to sort*/
                INFIELD = '';
                GOTO ENDE15;
             END;

             GET FILE (SYSIN) EDIT (INFIELD) (A(80));
             PUT SKIP EDIT (INFIELD)(A);
             CALL PLIRETC(12);  /*Input to sort continues*/
   ENDE15:
            RETURN(INFIELD);
             END E15X;

   E35X:   /* Output handling routine prints the sorted records*/
        PROC (INREC);

             DCL INREC CHAR(80);
             PUT SKIP EDIT (INREC) (A);
      NEXT:  CALL PLIRETC(4); /* Request next record from sort*/
             END E35X;
  END EX109;
 /*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
```

Figure 59. PLISRTD—sorting from input handling routine to output handling routine

Sorting variable-length records example

This example shows a program that sorts varying length records.

```
//OPT14#11 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
  /* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH
     RECORDS */

  EX1306:  PROC OPTIONS(MAIN);
          DCL RETURN_CODE FIXED BIN(31,0);
          CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
                       ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                       /*NOTE THAT LENGTH IS MAX AND INCLUDES
                         4 BYTE LENGTH PREFIX*/
                       1048576
                       RETURN_CODE,
                       PUTIN,
                       PUTOUT);

               SELECT(RETURN_CODE);
                 WHEN(0)  PUT SKIP EDIT (
                        'SORT COMPLETE RETURN_CODE 0') (A);
                 WHEN(16) PUT SKIP EDIT (
                        'SORT FAILED, RETURN_CODE 16') (A);
                 WHEN(20) PUT SKIP EDIT (
                        'SORT MESSAGE DATASET MISSING ') (A);
                 OTHER    PUT SKIP EDIT (
                      'INVALID RETURN_CODE = ', RETURN_CODE)
                         (A,F(2));
               END /* SELECT */;

     CALL PLIRETC(RETURN_CODE);
      /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
      PUTIN: PROC RETURNS (CHAR(80) VARYING);
        /*OUTPUT HANDLING ROUTINE*/
        /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE
          WHEN USING VARYING LENGTH RECORDS*/
          DCL STRING CHAR(80) VAR;

          ON ENDFILE(SYSIN) BEGIN;
               PUT SKIP EDIT ('END OF INPUT')(A);
               CALL PLIRETC(8);
               STRING = '';
               GOTO ENDPUT;
               END;

          GET EDIT(STRING)(A(80));
          I=INDEX(STRING||'  ','  ')-1;/*RESET LENGTH OF THE*/
          STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO */
                                       /* LENGTH OF TEXT IN */
                                       /* EACH INPUT RECORD.*/
```

Figure 60. Sorting varying-length records using input and output handling routines

```
          PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
          CALL PLIRETC(12);
 ENDPUT:  RETURN(STRING);
          END;
 PUTOUT:PROC(STRING);
         /*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
          DCL STRING CHAR (*);
           /*NOTE THAT FOR VARYING RECORDS THE STRING
             PARAMETER FOR THE OUTPUT-HANDLING ROUTINE
             SHOULD BE DECLARED ADJUSTABLE BUT CANNOT BE
             DECLARED VARYING*/
          PUT SKIP EDIT(STRING)(A);  /*PRINT THE SORTED DATA*/
          CALL PLIRETC(4);
          END;  /*ENDS PUTOUT*/
         END;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//*
```

Sorting varying-length records using input and output handling routines (continued)

Chapter 17. ILC with C

This chapter describes some aspects of InterLanguage Communication (ILC) between PL/I and C. The examples illustrate how to use many of the data types common to both languages and help you write PL/I code that either calls or is called by C.

Equivalent data types

This topic lists the common C and PL/I data type equivalents.

Table 35. C and PL/I type equivalents

C type	Matching PL/I type
char[...]	char(...) varyingz
wchar[...]	wchar(...) varyingz
signed char	fixed bin(7)
unsigned char	unsigned fixed bin(8)
short	fixed bin(15)
unsigned short	unsigned fixed bin(16)
int	fixed bin(31)
unsigned int	unsigned fixed bin(32)
long long	fixed bin(63)
unsigned long long	unsigned fixed bin(64)
float	float bin(21)
double	float bin(53)
long double	float bin(p) (p >= 54)
enum	ordinal
typedef	define alias
struct	define struct
union	define union
struct *	handle

Simple type equivalence

This example illustrates the translation of the simple `typedef` for `time_t` from the C header file `time.h`.

```
typedef long time_t;

define alias time_t fixed bin(31);
```

Figure 61. Simple type equivalence

© Copyright IBM Corp. 1999, 2017

Struct type equivalence

This example illustrates the translation of the simple `struct` for `tm` from the C header file `time.h`.

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};

define structure
  1 tm
     ,2 tm_sec   fixed bin(31)
     ,2 tm_min   fixed bin(31)
     ,2 tm_hour  fixed bin(31)
     ,2 tm_mday  fixed bin(31)
     ,2 tm_mon   fixed bin(31)
     ,2 tm_year  fixed bin(31)
     ,2 tm_wday  fixed bin(31)
     ,2 tm_yday  fixed bin(31)
     ,2 tm_isdst fixed bin(31)
   ;
```

Figure 62. Sample struct type equivalence

Enum type equivalence

This example illustrates the translation of the simple enum `__device_t` from the C header file `stdio.h`.

```
typedef enum {
   __disk       =  0,
   __terminal   =  1,
   __printer    =  2,
   __tape       =  3,
   __tdq        =  5,
   __dummy      =  6,
   __memory     =  8,
   __hfs        =  9,
   __hiperspace = 10
} __device_t;

define ordinal __device_t (
        __disk       value(0)
      , __terminal   value(1)
      , __printer    value(2)
      , __tape       value(3)
      , __tdq        value(4)
      , __dummy      value(5)
      , __memory     value(8)
      , __hfs        value(9)
      , __hiperspace value(10)
   );
```

Figure 63. Sample enum type equivalence

File type equivalence

A C file declaration depends on the platform, but it often starts as follows:

```
struct __file {
    unsigned char  *__bufPtr;
 ... } FILE;
```

Figure 64. Start of the C declaration for its FILE type

What is needed is a pointer (or token) for a file, so this translation can be finessed as follows:

```
define struct    1 file;
define alias     file_Handle  handle file;
```

Figure 65. PL/I equivalent for a C file

Using C functions

Suppose the programmer wants to write a program to read a file and dump it as formatted hex, by using the C functions **fopen** and **fread**.

The code for this program is straightforward:

```
filedump:
  proc(fn) options(noexecops main);

  dcl fn             char(*) var;

  %include filedump;

  file = fopen( fn, 'rb' );

  if file = sysnull() then
    do;
      display( 'file could not be opened' );
      return;
    end;

  do forever;
    unspec(buffer) = ''b;

    read_In = fread( addr(buffer), 1, stg(buffer), file );

    if read_In = 0 then
      leave;

    display(    heximage(addr(buffer),16,' ')  ||  '  '
             || translate(buffer,(32)'.',unprintable)  );

    if read_In < stg(buffer) then
      leave;
  end;

  call fclose( file );
end filedump;
```

Figure 66. Sample code to use fopen and fread to dump a file

Most of the declarations in the INCLUDE file `filedump` are obvious:

```
  define struct      1 file;
  define alias       file_Handle  handle file;

  define alias       size_t unsigned fixed bin(32);
  define alias       int signed fixed bin(31);

  dcl file           type(file_Handle);
  dcl read_In        fixed bin(31);
  dcl buffer         char(16);

  dcl unprintable    char(32) value( substr(collate(),1,32) );
```

Figure 67. Declarations for filedump program

Matching simple parameter types

It would be easy to mistranslate the declarations for the C functions. For instance, the declaration for the C function **fread** shown in Figure 68 on page 371 can be translated to the declaration as shown in Figure 69 on page 371.

```
size_t   fread(  void *,
                 size_t,
                 size_t,
                 FILE *);
```

Figure 68. C declaration of fread

```
dcl fread     ext
              entry( pointer,
                     type size_t,
                     type size_t,
                     type file_Handle )
              returns( type size_t );
```

Figure 69. First incorrect declaration of fread

On some platforms, this would not link successfully because C names are case sensitive. In order to prevent this kind of linker problem, it is best to specify the name in mixed case by using the extended form of the `external` attribute. So, for instance, the declaration for **`fread`** would be better as follows:

```
dcl fread     ext('fread')
              entry( pointer,
                     type size_t,
                     type size_t,
                     type file_Handle )
              returns( type size_t );
```

Figure 70. Second incorrect declaration of fread

But this would not run right, because while PL/I parameters are `byaddr` by default, C parameters are `byvalue` by default. To fix this issue, add the `byvalue` attribute to the parameters:

```
dcl fread     ext('fread')
              entry( pointer byvalue,
                     type size_t byvalue,
                     type size_t byvalue,
                     type file_Handle byvalue )
              returns( type size_t );
```

Figure 71. Third incorrect declaration of fread

But note how the return value is set in Figure 72 on page 372: a fourth parameter (the address of the temporary `_temp5`) is passed to the function **`fread`**, which is then expected to place the return code in the integer at that address. This is the convention for how values are returned when the `byaddr` attribute applies to `returns`, and PL/I uses this convention by default.

```
*      read_In = fread( addr(buffer), 1, stg(buffer), file );
*
       L     r4,FILE(,r13,176)
       L     r1,fread(,r5,12)
       LA    r2,_temp5(,r13,420)
       LA    r8,BUFFER(,r13,184)
       L     r15,&EPA_&WSA(,r1,8)
       L     r0,&EPA_&WSA(,r1,12)
       ST    r0,_CEECAA_(,r12,500)
       LA    r1,#MX_TEMP1(,r13,152)
       ST    r8,#MX_TEMP1(,r13,152)
       LA    r8,1
       ST    r8,#MX_TEMP1(,r13,156)
       ST    r7,#MX_TEMP1(,r13,160)
       ST    r4,#MX_TEMP1(,r13,164)
       ST    r2,#MX_TEMP1(,r13,168)
       BALR  r14,r15
       L     r0,_temp5(,r13,420)
       ST    r0,READ_IN(,r13,180)
```

Figure 72. Code generated for RETURNS BYADDR

This would not run right, because C return values are `byvalue`. To fix this error, add one more `byvalue` attribute.

```
dcl fread     ext('fread')
              entry( pointer byvalue,
                     type size_t byvalue,
                     type size_t byvalue,
                     type file_Handle byvalue )
              returns( type size_t byvalue );
```

Figure 73. Correct declaration of fread

Note how the return value is set now in Figure 74: no extra address is passed, and the return value is simply returned in register 15.

```
*      read_In = fread( addr(buffer), 1, stg(buffer), file );
*
       L     r2,FILE(,r13,176)
       L     r1,fread(,r5,12)
       LA    r7,BUFFER(,r13,184)
       L     r15,&EPA_&WSA(,r1,8)
       L     r0,&EPA_&WSA(,r1,12)
       ST    r0,_CEECAA_(,r12,500)
       LA    r1,#MX_TEMP1(,r13,152)
       ST    r7,#MX_TEMP1(,r13,152)
       LA    r7,1
       ST    r7,#MX_TEMP1(,r13,156)
       ST    r4,#MX_TEMP1(,r13,160)
       ST    r2,#MX_TEMP1(,r13,164)
       BALR  r14,r15
       LR    r0,r15
       ST    r0,READ_IN(,r13,180)
```

Figure 74. Code generated for RETURNS BYVALUE

Matching string parameter types

Now that **fread** is translated correctly, suppose a programmer tries this translation for **fopen**:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byvalue,
                    char(*) varyingz byvalue )
             returns( byvalue type file_handle );
```

Figure 75. First incorrect declaration of fopen

But C has no strings, only pointers, and these pointers would be passed by value (`byvalue`); so the strings should be by reference (`byaddr`):

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
             returns( byvalue type file_handle );
```

Figure 76. Second incorrect declaration of fopen

But PL/I passes descriptors with strings and C does not understand them, so they must be suppressed. To suppress the descriptors, add `options(nodescriptor)` to the declaration:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
             returns( byvalue type file_handle )
             options ( nodescriptor );
```

Figure 77. Correct declaration of fopen

This will work, but is not optimal because the parameters are input-only; if the parameter is a constant, the `nonassignable` attribute will prevent a copy being made and passed. Hence, the best translation of the declaration of **fopen** is as follows:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz nonasgn byaddr,
                    char(*) varyingz nonasgn byaddr )
             returns( byvalue type file_handle )
             options ( nodescriptor );
```

Figure 78. Optimal, correct declaration of fopen

At this point, the declare for the **fclose** function presents few surprises except perhaps for the `optional` attribute in the `returns` specification. This attribute allows you to invoke the **fclose** function through a CALL statement and not have to dispose of the return code. But note that if the file were an output file, the return code on **fclose** should always be checked because the last buffer is written

out only when the file is closed and that write could fail for lack of space.

```
dcl fclose   ext('fclose')
             entry( type file_handle byvalue )
             returns( optional type int byvalue )
             options ( nodescriptor );
```

Figure 79. Declaration of fclose

Now, on z/OS UNIX, you can compile and run the programs with the commands:

```
    pli -qdisplay=std filedump.pli

    filedump filedump.pli
```

Figure 80. Commands to compile and run filedump

This will produce the following output:

```
   15408689 938584A4 94977A40 97999683  . filedump: proc
   4D86955D 409697A3 899695A2 4D959685  (fn) options(noe
   A7858396 97A24094 8189955D 5E151540  xecops main);..
```

Figure 81. Output of running filedump

Functions returning ENTRYs

The C quicksort function **qsort** takes a compare routine. For instance, to sort an array of integers, the following function (which use the `byvalue` attribute twice) could be used:

```
comp2:
  proc( key, element )
  returns( fixed bin(31) byvalue );

  dcl (key, element) pointer byvalue;
  dcl word based fixed bin(31);

  select;
    when( key->word < element->word )
      return( -1 );
    when( key->word = element->word )
      return( 0 );
    otherwise
      return( +1 );
  end;
end;
```

Figure 82. Sample compare routine for C qsort function

And the C **qsort** function could be used with this compare routine to sort an array of integers, as in the following code fragment:

```
dcl a(1:4) fixed bin(31) init(19,17,13,11);

put skip data( a );

call qsort( addr(a), dim(a), stg(a(1)), comp2 );

put skip data( a );
```

Figure 83. Sample code to use C qsort function

But because C function pointers are not the same as PL/I ENTRY variables, the C **qsort** function must not be declared simply as follows:

```
dcl qsort    ext('qsort')
             entry( pointer,
                    fixed bin(31),
                    fixed bin(31),
                    entry returns( byvalue fixed bin(31) )
                  )
             options( byvalue nodescriptor );
```

Figure 84. Incorrect declaration of qsort

Recall that a PL/I ENTRY variable might point to a nested function (and thus requires a backchain address as well as an entry point address). But a C function pointer is limited in pointing to a non-nested function only, and so a PL/I ENTRY variable and a C function pointer do not even use the amount of storage.

However, a C function pointer is equivalent to the PL/I type LIMITED ENTRY. Therefore, the C **qsort** function could be declared as follows:

```
dcl qsort    ext('qsort')
             entry( pointer,
                    fixed bin(31),
                    fixed bin(31),
                    limited entry
                    returns( byvalue fixed bin(31) )
                  )
             options( byvalue nodescriptor );
```

Figure 85. Correct declaration of qsort

Linkages

On z/OS, there are two crucial facts about linkages:

- IBM C, JAVA and Enterprise PL/I use the same linkage by default.
- This linkage is not the system linkage.

For a traditional PL/I application where all parameters are `byaddr`, the differences between the code generated when a function has the default linkage and the code generated when the function has the system linkage would usually not matter. But if the parameters are `byvalue` (as they usually are in C and JAVA), the differences can break your code.

In fact, there is only a small difference if the parameters are byaddr. In Figure 86, the only difference between the code generated for a function with the default linkage and for one with the system linkage is that the high-order bit is turned on for the last parameter of the system linkage call.

This difference is transparent to most programs.

```
dcl dfta  ext entry( fixed bin(31) byaddr
                   ,fixed bin(31) byaddr );
dcl sysa  ext entry( fixed bin(31) byaddr
                   ,fixed bin(31) byaddr )
           options( linkage(system) );

*      call dfta( n, m );
*
          LA    r0,M(,r13,172)
          LA    r2,N(,r13,168)
          L     r15,=V(DFTV)(,r3,126)
          LA    r1,#MX_TEMP1(,r13,152)
          ST    r2,#MX_TEMP1(,r13,152)
          ST    r0,#MX_TEMP1(,r13,156)
          BALR  r14,r15
*
*      call sysa( n, m );
*
          LA    r0,M(,r13,172)
          LA    r2,N(,r13,168)
          O     r0,=X'80000000'
          L     r15,=V(SYSV)(,r3,130)
          LA    r1,#MX_TEMP1(,r13,152)
          ST    r2,#MX_TEMP1(,r13,152)
          ST    r0,#MX_TEMP1(,r13,156)
          BALR  r14,r15
```

Figure 86. Code when parameters are BYADDR

But, there is a big difference if the parameters are byvalue rather than byaddr. In Figure 87 on page 377, for the function with the default linkage, register 1 points to the values of the integers passed, while for the function with the system linkage, register 1 points to the addresses of those values.

This difference is not transparent to most programs.

```
dcl dftv  ext entry( fixed bin(31) byvalue
                    ,fixed bin(31) byvalue );
dcl sysv  ext entry( fixed bin(31) byvalue
                    ,fixed bin(31) byvalue )
           options( linkage(system) );

*      call dftv( n, m );
*
          L     r2,N(,r13,168)
          L     r0,M(,r13,172)
          L     r15,=V(DFTV)(,r3,174)
          LA    r1,#MX_TEMP1(,r13,152)
          ST    r2,#MX_TEMP1(,r13,152)
          ST    r0,#MX_TEMP1(,r13,156)
          BALR  r14,r15
*
*      call sysv( n, m );
*
          L     r1,N(,r13,168)
          L     r0,M(,r13,172)
          ST    r0,#wtemp_1(,r13,176)
          LA    r0,#wtemp_1(,r13,176)
          ST    r1,#wtemp_2(,r13,180)
          LA    r2,#wtemp_2(,r13,180)
          O     r0,=X'80000000'
          L     r15,=V(SYSV)(,r3,178)
          LA    r1,#MX_TEMP1(,r13,152)
          ST    r2,#MX_TEMP1(,r13,152)
          ST    r0,#MX_TEMP1(,r13,156)
          BALR  r14,r15
```

Figure 87. Code when parameters are BYVALUE

Sharing output and input

This section provides information about sharing output and input with a C program by specifying the STDSYS option.

For more information and limitations about the STDSYS option, see the following information:

- "STDSYS" on page 84 in Chapter 1, "Using compiler options and facilities," on page 3.
- Stream I/O with unprintable characters in *Enterprise PL/I for z/OS Compiler and Run-Time Migration Guide*.

Sharing output

If you want to share SYSPRINT with a C program, you must compile your PL/I code with the STDSYS option.

By default, DISPLAY statements use WTO's to display their output. If you specify the DISPLAY(STD) compiler option, DISPLAY statements will use the C **puts** function to display their output. This can be particularly useful under z/OS UNIX.

Behavior of the standard C stream for sharing output under MVS batch, TSO batch, IMS batch, and IMS interactive is as follows:

1. `stdout` goes first to DD:SYSPRINT.

2. If DD:SYSPRINT does not exist, stdout looks for DD:SYSTERM.
3. If neither DD:SYSTERM nor DD:SYSERR exists, the library opens a sysout=* data set by using DD SYSPRINT and sends the stdout stream to it.

Sharing input

To share SYSIN with a C program, you must compile the application with the STDSYS option and open SYSIN as an input stream file. Avoid using the DD names that are reserved by the C Library.

You can also copy SYSIN to a temporary data set in a prior job step and use that as SYSIN in your PL/I job step; it can be shared when it is not allocated to an instream file.

An input stream file can be opened only once when it is allocated to the SYSIN DD in JCL.

Behavior of the standard C stream for sharing input under MVS batch, TSO batch, IMS batch, and IMS interactive is as follows:

1. stdin goes to DD:SYSIN.
2. If DD:SYSIN does not exist, all read operations from stdin fails.

Using the ATTACH statement

The ATTACH statement in a PL/I program uses the underlying pthread library to perform thread management. When the ATTACH statement is issued, the pthread library tries to allocate standard C stream files.

After the ATTACH statement is issued, any attempt to open SYSIN by the application fails when SYSIN is an instream file that is allocated to the SYSIN DD in JCL (for example, //SYSIN DD * or //SYSIN DD DATA). A SYSIN OPEN FAILED error is issued.

Redirecting C standard streams

Use the STDSYS option in all mixed PL/I and C applications and all multithreading applications.

If you compile your program with the NOSTDSYS option, it might conflict with the use of the SYSIN DD name and the SYSPRINT DD name in C.

For example, if your program has the ATTACH statement, it starts the C environment directly. Starting the C environment causes the SYSIN and SYSPRINT streams to be opened and closed independent of your PL/I application programs. In same cases, SYSIN might fail to open, or the SYSPRINT data set might be overwritten.

Summary

This topic provides a summary of the key points described in this section.

- C is case sensitive.
- Parameters should be BYVALUE.
- Return values should be BYVALUE.
- String parameters should be BYADDR.
- Arrays and structures should also be BYADDR.

- No descriptors should be passed.
- Input-only strings should be NONASSIGNABLE.
- C function pointers map to LIMITED ENTRYs.
- The IBM C compilers and the IBM PL/I compilers use the same default linkage (and it matters).

Chapter 18. Interfacing with Java

This chapter gives a brief description of Java™ and the Java Native Interface (JNI) and explains why you might be interested in using it with PL/I.

This chapter describes a simple Java-PL/I application and provides information about compatibility between the two languages. Instructions on how to build and run the Java-PL/I sample applications assume that the work is being done in the z/OS UNIX System Services environment of z/OS.

Before you can communicate with Java from PL/I, you need to have Java installed on your z/OS system. Contact your local System Administrator for more information about how to set up your z/OS Java environment.

These sample programs have been compiled and tested with Java JRE Version 1.6.0. To determine the level of Java in your z/OS UNIX System Services environment, enter this command from the command line:

```
java -version
```

The active Java version is then displayed, as in the following example:

```
java version "1.6.0"
Java(TM) SE Runtime Environment (build pmz3160_26sr1-20111114_01(SR1))
IBM J9 VM (build 2.6, JRE 1.6.0 z/OS s390-31 20111113_94967 (JIT enabled, AOT enabled)
```

Java Native Interface (JNI)

Java is an object-oriented programming language invented by Sun Microsystems and provides a powerful way to make Internet documents interactive. The Java Native Interface (JNI) is the Java interface to native programming languages and is part of the Java Development Kits.

By writing programs that use JNI, you can ensure that your code is portable across many platforms.

The JNI allows Java code that runs within a Java Virtual Machine (JVM) to operate with applications and libraries written in other languages, such as PL/I. In addition, the *Invocation API* allows you to embed a Java Virtual Machine into your PL/I applications.

Java is a fairly complete programming language; however, there are situations in which you want to call a program written in another programming language. You can do this from Java with a method call to a native language, known as a *native method*.

These are some reasons to use native methods:

- The native language has a special capability that your application needs and that the standard Java class libraries lack.
- You already have many existing applications in your native language and you wish to make them accessible to a Java application.
- You want to implement an intensive series of complicated calculations in your native language and have your Java applications call these functions.

© Copyright IBM Corp. 1999, 2017

- You or your programmers have a broader skill set in your native language and you do not wish to loose this advantage.

Programming through the JNI lets you use native methods to do many different operations:

- A native method can utilize Java objects in the same way that a Java method uses these objects.
- A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks.
- A native method can inspect and use objects created by Java application code.
- A native method can update Java objects that it created or were passed to it, and these updated objects can then be made available to the Java application.

Finally, native methods can also easily call already existing Java methods, capitalizing on the functionality already incorporated in the Java programming framework. In these ways, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

Calling PL/I program from Java

When a PL/I program is called from a Java program, all files opened in PL/I must be explicitly closed in PL/I before PL/I returns control to Java for the last time.

Similarly, all modules that are fetched in the PL/I program must be released.

JNI sample program #1 - 'Hello World'

The first sample program is another variation of the "Hello World!" program. The "Hello World!" program has one Java class, `callingPLI.java`. The native method, written in PL/I, is contained in `hiFromPLI.pli`.

Here is a brief overview of the steps to create this sample program:

1. Write a Java program that defines a class containing a native method, loads the native load library, and calls the native method.
2. Compile the Java program to create a Java class.
3. Write a PL/I program that implements the native method and displays the "Hello!" text.
4. Compile and link the PL/I program.
5. Run the Java program that calls the native method in the PL/I program.

Step 1: Writing the Java program

Procedure

1. Declare the native method.

   All methods, whether Java methods or native methods, must be declared within a Java class. The only difference in the declaration of a Java method and a native method is the keyword **`native`**. The **`native`** keyword tells Java that the implementation of this method will be found in a native library that will be loaded during the execution of the program. You can declare the native method as follows:

   ```
   public native void callToPLI();
   ```

In the above statement, the `void` means that there is no return value expected from this native method call. The empty parentheses in the method name `callToPLI( )` means that there are no parameters being passed on the call to the native method.

2. Load the native library so that the native library will be loaded at execution time.

   You can use the following Java statement to load the native library:

   ```
   static {
     System.loadLibrary("hiFromPLI");
   }
   ```

   In the above statement, the Java System method `System.loadLibrary(...)` is called to find and load the native library. The PL/I shared library, `libhiFromPLI.so`, will be created during the step that compiles and links the PL/I program.

3. Write the Java Main method.

   The `callingPLI` class also includes a `main` method to instantiate the class and call the native method. The `main` method instantiates `callingPLI` and calls the `callToPLI()` native method.

   The complete definition of the `callingPLI` class, including all the points addressed above in this topic, is as follows:

   ```
   public class callingPLI {
     public native void callToPLI();
     static {
        System.loadLibrary("hiFromPLI");
     }
     public static void main(String[] argv) {
        callingPLI callPLI = new callingPLI();
        callPLI.callToPLI();
        System.out.println("And Hello from Java, too!");
     }
   }
   ```

Step 2: Compiling the Java program

Procedure

Use the Java compiler to compile the `callingPLI` class into an executable form. You can use the following command:

```
javac callingPLI.java
```

Step 3: Writing the PL/I Program

The PL/I implementation of the native method looks much like any other PL/I subroutine.

Useful PL/I compiler options

The sample program contains a series of `*PROCESS` statements that define the important compiler options.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
```

Here is a brief description of them and why they are useful:

Extname(100)
Allows for longer, Java style, external names.

Margins(1,100)
Extending the margins gives you more room for Java style names and identifiers.

Display(Std)
Writes the "Hello World" text to stdout, not through WTOs. In the z/OS UNIX environment WTOs would not be seen by the user.

Dllinit
Includes the initialization code needed for creating a DLL.

Extrn(Short)
EXTRNs are emitted only for those constants that are referenced. This option is necessary for Enterprise PL/I V3R3 and later.

Default(ASCII IEEE);
ASCII specifies that CHARACTER and PICTURE data is held in ASCII - the form in which it is held by JAVA.

IEEE specifies that FLOAT data is held in IEEE format - the form in which it is held by JAVA.

RENT
Ensures that code is reentrant even if it writes on static variables.

Correct form of PL/I procedure name and procedure statement

The PL/I procedure name must conform to the Java naming convention in order to be located by the Java Class Loader at execution time. The Java naming scheme consists of three parts. The first part identifies the routine to the Java environment, the second part is the name of the Java class that defines the native method, and the third part is the name of the native method itself.

Here is a breakdown of PL/I procedure name `Java_callingPLI_callToPLI` in the sample program:

Java
All native methods resident in dynamic libraries must begin with `Java`.

_callingPLI
The name of the Java class that declares the native method.

_callToPLI
The name of the native method itself.

Note: There is an important difference between coding a native method in PL/I and in C. The **javah** tool, which is supplied with the JDK, generates the form of the external references required for C programs. When you write your native methods in PL/I and follow the rules above for naming your PL/I external references, performing the **javah** step is not necessary for PL/I native methods.

In addition, the following options must be specified in the OPTIONS option on the PROCEDURE statement:

- FromAlien
- NoDescriptor
- ByValue

The complete procedure statement for the sample program is as follows:

```
Java_callingPLI_callToPLI:
Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI include file

The two PL/I include files that contain the PL/I definitions of the Java Native interface are `ibmzjni.inc`, which in turn includes `ibmzjnim.inc`. These include files are included with this statement:

```
%include ibmzjni;
```

The `ibmzjni` and `ibmzjnim` include files are provided in the PL/I SIBMZSAM data set.

The complete PL/I procedure

For completeness, here is the entire PL/I program that defines the native method:

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
PliJava_Demo: Package Exports(*);

 Java_callingPLI_callToPLI:
 Proc( JNIEnv , MyJObject )
   External( "Java_callingPLI_callToPLI" )
   Options( FromAlien NoDescriptor ByValue );

  %include ibmzjni;
  Dcl myJObject        Type jObject;

  Display('Hello from Enterprise PL/I!');

End;
```

Step 4: Compiling and linking the PL/I program

Procedure

1. Compile the PL/I sample program with the following command:

   ```
   pli -c hiFromPLI.pli
   ```

2. Link the resulting PL/I object deck into a shared library with this command:

   ```
   c89 -o libhiFromPLI.so hiFromPLI.o
   ```

   Ensure to include the `lib` prefix on the name of the PL/I shared library; otherwise, the Java class loader cannot find it.

Step 5: Running the sample program

Procedure

Run the Java-PL/I sample program with this command:

```
java callingPLI
```

The output of the sample program is as follows:

```
Hello from Enterprise PL/I!
And Hello from Java, too!
```

The first line written from the PL/I native method. The second line is from the calling Java class after returning from the PL/I native method call.

JNI sample program #2 - Passing a string

This sample program passes a string back and forth between Java and PL/I.

See Figure 88 on page 387 for the complete listing of the `jPassString.java` program. The Java portion has one Java class, `jPassString.java`. The native method, written in PL/I, is contained in `passString.pli`. Much of the information from "JNI sample program #1 - 'Hello World'" on page 382 applies to this sample program as well. The following topics discuss only new or different aspects for this sample program.

Step 1: Writing the Java program

Procedure

1. Declare the native method.

   ```
   public native void pliShowString();
   ```

2. Load the native library.

   ```
   static {
     System.loadLibrary("passString");
   }
   ```

3. Write the Java Main method.

   The `jPassString` class also includes a `main` method to instantiate the class and call the native method. The `main` method instantiates `jPassString` and calls the `pliShowString()` native method.

   This sample program prompts the user for a string and reads that value in from the command line. This is done within a `try/catch` statement as shown in Figure 88 on page 387.

```
// Read a string, call PL/I, display new string upon return
import java.io.*;

public class jPassString{

 /* Field to hold Java string                          */
 String myString;

 /* Load the PL/I native library                       */
 static {
        System.loadLibrary("passString");
 }

 /* Declare the PL/I native method                     */
 public native void pliShowString();

 /* Main Java class                                    */
 public static void main(String[] arg) {

   System.out.println(" ");

   /* Instantiate Java class and initialize string     */
   jPassString myPassString = new jPassString();
   myPassString.myString = " ";

   /* Prompt user for a string                         */
   try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));

        /* Process until 'quit' received               */
        while (!myPassString.myString.equalsIgnoreCase("quit")) {
          System.out.println(
            "From Java: Enter a string or 'quit' to quit.");
          System.out.print("Java Prompt > ");
          /* Get string from command line              */
          myPassString.myString = in.readLine();
          if (!myPassString.myString.equalsIgnoreCase("quit"))
            {
             /* Call PL/I native method                */
             myPassString.pliShowString();
             /* Return from PL/I and display new string */
             System.out.println(" ");
             System.out.println(
                 "From Java: String set by PL/I is: "
               + myPassString.myString );
            }
        }
        }  catch (IOException e) {
        }
 }
}
```

Figure 88. Java sample program #2 - Passing a string

Step 2: Compiling the Java program

Procedure

Use the Java compiler to compile the Java code. You can use the following command:

```
javac jPassString.java
```

Step 3: Writing the PL/I program

All of the information about writing the PL/I "Hello World" sample program, as described in “Step 3: Writing the PL/I Program” on page 383, applies to this program as well.

Correct form of PL/I procedure name and procedure statement

The PL/I procedure name for this program is `Java_jPassString_pliShowString`.

The complete procedure statement for the sample program is as follows:

```
Java_jPassString_pliShowString:
Proc( JNIEnv , myjobject )
  external( "Java_jPassString_pliShowString" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI include file

The two PL/I include files that contain the PL/I definitions of the Java Native interface are `ibmzjni.inc`, which in turn includes `ibmzjnim.inc`. These include files are included with this statement:

```
%include ibmzjni;
```

The `ibmzjni` and `ibmzjnim` include files are provided in the PL/I SIBMZSAM data set.

The complete PL/I procedure

The complete PL/I program is shown in Figure 89 on page 389. This sample PL/I program makes several calls through JNI.

Upon entry, a reference to the calling Java Object, `myObject` is passed into the PL/I procedure. The PL/I program uses this reference to get information from the calling object. The first piece of information is the Class of the calling object, which is retrieved through the **`GetObjectClass`** JNI function. This Class value is then used by the **`GetFieldID`** JNI function to get the identity of the Java string field in the Java object. This Java field is further identified by the name of the field, `myString`, and the JNI field descriptor, `Ljava/lang/String;`, which identifies the field as a Java String field. The value of the Java string field is then retrieved by the **`GetObjectField`** JNI function. Before PL/I can use the Java string value, it must be unpacked into a form that PL/I can understand. The **`GetStringUTFChars`** JNI function converts the Java string into a PL/I varyingz string, which is then displayed by the PL/I program.

After displaying the retrieved Java string, the PL/I program prompts the user for a PL/I string to be used to update the string field in the calling Java object. The PL/I string value is converted to a Java string by the **`NewString`** JNI function. This new Java string is then used to update the string field in the calling Java object by the **`SetObjectField`** JNI function.

When the PL/I program ends, control is returned to Java, where the newly updated Java string is displayed by the Java program.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
plijava_demo: package exports(*);

 Java_passString_pliShowString:
 Proc( JNIEnv , myJObject )
   external( "Java_jPassString_pliShowString" )
   Options( FromAlien NoDescriptor ByValue );

  %include ibmzjni;

  Dcl myBool           Type jBoolean;
  Dcl myClazz          Type jClass;
  Dcl myFID            Type jFieldID;
  Dcl myJObject        Type jObject;
  Dcl myJString        Type jString;
  Dcl newJString       Type jString;
  Dcl myID             Char(9)   Varz static init( 'myString' );
  Dcl mySig            Char(18)  Varz static
                       init( 'Ljava/lang/String;' );
  Dcl pliStr           Char(132) Varz Based(pliStrPtr);
  Dcl pliReply         Char(132) Varz;
  Dcl pliStrPtr        Pointer;
  Dcl nullPtr          Pointer;

  Display('  ');

  /* Get information about the calling Class             */
  myClazz = GetObjectClass(JNIEnv, myJObject);

  /* Get Field ID for String field from Java             */
  myFID = GetFieldID(JNIEnv, myClazz, myID, mySig );

  /* Get the Java String in the string field             */
  myJString = GetObjectField(JNIEnv, myJObject, myFID );

  /* Convert the Java String to a PL/I string            */
  pliStrPtr = GetStringUTFChars(JNIEnv, myJString, myBool );

  Display('From  PLI: String retrieved from Java is: ' || pliStr );
  Display('From  PLI: Enter a string to be returned to Java:' )
         reply(pliReply);

  /* Convert the new PL/I string to a Java String        */
  newJString = NewString(JNIEnv, trim(pliReply), length(pliReply) );

  /* Change the Java String field to the new string value */
  nullPtr = SetObjectField(JNIEnv, myJObject, myFID, newJString);

 End;

end;
```

Figure 89. PL/I sample program #2 - Passing a string

Step 4: Compiling and linking the PL/I program

Procedure

1. Compile the PL/I sample program with the following command:

   ```
   pli -c passString.pli
   ```

2. Link the resulting PL/I object deck into a shared library with this command:

   ```
   c89 -o libpassString.so passString.o
   ```

   Ensure to include the `lib` prefix on the name; otherwise, the PL/I shared library or the Java class loader cannot find it.

Step 5: Running the sample program

Procedure

Run the Java-PL/I sample program with this command:

```
java jPassString
```

The output of the sample program, complete with the prompts for user input from both Java and PL/I, is as follows:

```
>java jPassString

From Java: Enter a string or 'quit' to quit.
Java Prompt > A string entered in Java

From  PLI: String retrieved from Java is: A string entered in Java
From  PLI: Enter a string to be returned to Java:
A string entered in PL/I

From Java: String set by PL/I is: A string entered in PL/I
From Java: Enter a string or 'quit' to quit.
Java Prompt > quit
>
```

JNI sample program #3 - Passing an integer

This sample program passes an integer back and forth between Java and PL/I.

See Figure 90 on page 392 for the complete listing of the `jPassInt.java` program. The Java portion has one Java class, `jPassInt.java`. The native method, written in PL/I, is contained in `passInt.pli`. Much of the information from "JNI sample program #1 - 'Hello World'" on page 382 applies to this sample program as well. The following topics discuss only new or different aspects for this sample program.

Step 1: Writing the Java program

Procedure

1. Declare the native method.

   ```
   public native void pliShowInt();
   ```

2. Load the native library.

   ```
   static {
     System.loadLibrary("passInt");
   }
   ```

3. Write the Java Main method.

The `jPassInt` class also includes a `main` method to instantiate the class and call the native method. The `main` method instantiates `jPassInt` and calls the `pliShowInt()` native method.

This sample program prompts the user for an integer and reads that value in from the command line. This is done within a `try/catch` statement as shown in Figure 90 on page 392.

```
// Read an integer, call PL/I, display new integer upon return
import java.io.*;
import java.lang.*;

public class jPassInt{

 /* Fields to hold Java string and int                 */
 int myInt;
 String myString;

 /* Load the PL/I native library                       */
 static {
        System.loadLibrary("passInt");
 }

 /* Declare the PL/I native method                     */
 public native void pliShowInt();

 /* Main Java class                                    */
 public static void main(String[] arg) {

   System.out.println(" ");

   /* Instantiate Java class and initialize string     */
   jPassInt pInt = new jPassInt();
   pInt.myInt = 1024;
   pInt.myString = " ";

   /* Prompt user for an integer                       */
   try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));

        /* Process until 'quit' received               */
        while (!pInt.myString.equalsIgnoreCase("quit")) {
          System.out.println
            ("From Java: Enter an Integer or 'quit' to quit.");
          System.out.print("Java Prompt > ");
          /* Get string from command line              */
          pInt.myString = in.readLine();
          if (!pInt.myString.equalsIgnoreCase("quit"))
            {
             /* Set int to integer value of String     */
             pInt.myInt = Integer.parseInt( pInt.myString );
             /* Call PL/I native method                */
             pInt.pliShowInt();
             /* Return from PL/I and display new string */
             System.out.println(" ");
             System.out.println
               ("From Java: Integer set by PL/I is: " + pInt.myInt );
            }
        }
        }  catch (IOException e) {
        }
 }

}
```

Figure 90. Java sample program #3 - Passing an integer

Step 2: Compiling the Java program

Procedure

Use the Java compiler to compile the Java code. You can use the following command:

```
javac jPassInt.java
```

Step 3: Writing the PL/I program

All of the information about writing the PL/I "Hello World" sample program, as described in “Step 3: Writing the PL/I Program” on page 383, applies to this program as well.

Correct form of PL/I procedure name and procedure statement

The PL/I procedure name for this program is `Java_jPassInt_pliShowInt`.

The complete procedure statement for the sample program is as follows:

```
Java_passNum_pliShowInt:
Proc( JNIEnv , myjobject )
  external( "Java_jPassInt_pliShowInt" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI include file

The two PL/I include files that contain the PL/I definitions of the Java native interface are `ibmzjni.inc`, which in turn includes `ibmzjnim.inc`. These include files are included with this statement:

```
%include ibmzjni;
```

The `ibmzjni` and `ibmzjnim` include files are provided in the PL/I SIBMZSAM data set.

The complete PL/I procedure

The complete PL/I program is shown in Figure 91 on page 394. This sample PL/I program makes several calls through the JNI.

Upon entry, a reference to the calling Java object, `myObject`, is passed into the PL/I procedure. The PL/I program uses this reference to get information from the calling object. The first piece of information is the Class of the calling object, which is retrieved by the **`GetObjectClass`** JNI function. This Class value is then used by the **`GetFieldID`** JNI function to get the identity of the Java integer field in the Java object. This Java field is further identified by the name of the field, `myInt`, and the JNI field descriptor, `I`, which identifies the field as an integer field. The value of the Java integer field is then retrieved by the **`GetIntField`** JNI function, which is then displayed by the PL/I program.

After displaying the retrieved Java integer, the PL/I program prompts the user for a PL/I integer to be used to update the integer field in the calling Java object. The PL/I integer value is then used to update the integer field in the calling Java object by the **`SetIntField`** JNI function.

When the PL/I program ends, control is returned to Java, where the newly updated Java integer is displayed by the Java program.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
plijava_demo: package exports(*);

 Java_passNum_pliShowInt:
 Proc( JNIEnv , myjobject )
   external( "Java_jPassInt_pliShowInt" )
   Options( FromAlien NoDescriptor ByValue );

  %include ibmzjni;

  Dcl myClazz          Type jClass;
  Dcl myFID            Type jFieldID;
  Dcl myJInt           Type jInt;
  dcl rtnJInt          Type jInt;
  Dcl myJObject        Type jObject;
  Dcl pliReply         Char(132) Varz;
  Dcl nullPtr          Pointer;

  Display('  ');

  /* Get information about the calling Class             */
  myClazz = GetObjectClass(JNIEnv, myJObject);

  /* Get Field ID for int field from Java                */
  myFID = GetFieldID(JNIEnv, myClazz, "myInt", "I");

  /* Get Integer value from Java                         */
  myJInt = GetIntField(JNIEnv, myJObject, myFID);

  display('From  PLI: Integer retrieved from Java is: ' || trim(myJInt) );
  display('From  PLI: Enter an integer to be returned to Java:' )
         reply(pliReply);

  rtnJInt = pliReply;

  /* Set Integer value in Java from PL/I                 */
  nullPtr = SetIntField(JNIEnv, myJObject, myFID, rtnJInt);

 End;

end;
```

Figure 91. PL/I sample program #3 - Passing an integer

Step 4: Compiling and linking the PL/I program

Procedure

1. Compile the PL/I sample program with the following command:

   ```
   pli -c passInt.pli
   ```

2. Link the resulting PL/I object deck into a shared library with this command:

   ```
   c89 -o libpassInt.so passInt.o
   ```

   Ensure to include the `lib` prefix on the name; otherwise, the PL/I shared library or the Java class loader cannot find it.

Step 5: Running the sample program

Procedure

Run the Java-PL/I sample program with this command:

```
java jPassInt
```

The output of the sample program, complete with the prompts for user input from both Java and PL/I, is as follows:

```
>java jPassInt

From Java: Enter an Integer or 'quit' to quit.
Java Prompt > 12345

From  PLI: Integer retrieved from Java is: 12345
From  PLI: Enter an integer to be returned to Java:
54321

From Java: Integer set by PL/I is: 54321
From Java: Enter an Integer or 'quit' to quit.
Java Prompt > quit
>
```

JNI sample program #4 - Java invocation API

This sample program is a little different from the previous samples. In this sample, PL/I invokes Java first through the Java invocation API, creating an embedded Java Virtual Machine (JVM). PL/I then calls a Java method, passing to it a string that the Java method then displays.

The PL/I sample program is named `createJVM.pli` and the Java method it calls is contained in `javaPart.java`.

Step 1: Writing the Java program

About this task

Because this sample does not use a PL/I native method, there is no need to declare one. Instead, the Java portion for this sample is just a simple Java method.

Procedure

Write the Java Main method.
The `javaPart` class contains only one statement. This statement prints out a short `'Hello World...'` from Java, and then appends the string that was passed to it from the PL/I program. The entire class is shown in Figure 92.

```
// Receive a string from PL/I then display it after saying "Hello"
public class javaPart {
   public static void main(String[] args) {
      System.out.println("From Java  -  Hello World... " + args[0]);
   }
}
```

Figure 92. Java sample program #4 - Receiving and printing a string

Step 2: Compiling the Java program

Procedure

Use the Java compiler to compile the Java code. You can use the following command:

```
javac javaPart.java
```

Step 3: Writing the PL/I program

Most of the information about writing the PL/I "Hello World" sample program, as described in “Step 3: Writing the PL/I Program” on page 383, applies to this program as well. However, because in this sample PL/I is calling Java, there are some additional points to consider.

Correct form of PL/I procedure name and procedure statement

Because in this sample the PL/I program is calling Java, the PL/I program is MAIN. There is no need to be concerned about the external name of this PL/I program because it is not referenced.

The complete procedure statement for the sample program is as follows:

```
createJVM: Proc Options(Main);
```

JNI include file

The two PL/I include files that contain the PL/I definitions of the Java native interface are `ibmzjni.inc`, which in turn includes `ibmzjnim.inc`. Even though in this sample PL/I is calling Java, these include files are still necessary. These include files are included with this statement:

```
%include ibmzjni;
```

The `ibmzjni` and `ibmzjnim` include files are provided in the PL/I SIBMZSAM data set.

Linking the PL/I program with the Java library

Because this PL/I sample program calls Java, the program must link to the Java library. The Java libraries are linked with XPLINK and the PL/I modules are not. PL/I can still link to and call XPLINK libraries but you must use the PLIXOPT variable to specify the **XPLINK=ON** runtime option. You can declare the PLIXOPT variable as follows:

```
Dcl PLIXOPT     Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );
```

For a description of PLIXOPT, see the *z/OS Language Environment Programming Guide*.

Using the Java invocation API

This PL/I sample program calls the Java invocation API **JNI_CreateJavaVM** to create its own embedded JVM. This API requires certain structures to be set up and initialized correctly as shown in Figure 93 on page 398.

1. **JNI_GetDefaultJavaVMInitArgs** is called to get the default initilization options.

2. These default options are modified with the addition of the `java.class.path` information.
3. **`JNI_CreateJavaVM`** is called to create the embedded JVM.

The complete PL/I program

The complete PL/I program is shown in Figure 93 on page 398. This sample PL/I program makes several calls through the JNI.

In this sample, the reference to the Java object, a newly created JVM in this case, is not passed in but is instead returned from the call to the **`JNI_CreateJavaVM`** API. The PL/I program uses this reference to get information from the JVM. The first piece of information is the Class containing the Java method to call. This Class is found by the **`FindClass`** JNI function. The Class value is then used by the **`GetStaticMethodID`** JNI function to get the identity of the Java method that will be called.

Before calling this Java method, convert the PL/I string into a format that Java understands. The PL/I program holds the string in ASCII format. Java strings are stored in UTF format. In addition, Java strings are not really strings as PL/I programmers understand them but are themselves a Java class and can only be modified through methods. To create a Java string, use the **`NewStringUTF`** JNI function. This function returns a Java object called `myJString` that contains the PL/I string converted to UTF. Next create a Java object array by calling the **`NewObjectArray`** JNI function, passing to it the reference to the `myJString` object. This function returns a reference to a Java object array containing the string for the Java method to display.

Now the Java method can be called by the **`CallStaticVoidMethod`** JNI function and will then display the string passed to it. After displaying the string, the PL/I program destroys the embedded JVM by using the **`DestroyJavaVM`** JNI function and the PL/I program completes.

The complete source of the PL/I program is shown in Figure 93 on page 398.

```
*Process Limits( Extname( 100 ) )  Margins( 1, 100 );
*Process Margins( 1, 100 ) ;
*Process Display(STD) Rent;
*Process Default( ASCII ) Or('|');
createJVM: Proc Options(Main);

  %include ibmzjni;

  Dcl myJObjArray Type jobjectArray;
  Dcl myClass     Type jclass;
  Dcl myMethodID  Type jmethodID;
  Dcl myJString   Type jstring;
  Dcl myRC        Fixed Bin(31)  Init(0);
  Dcl myPLIStr    Char(50) Varz
                  Init('... a PLI string in a Java Virtual Machine!');
  Dcl OptStr1 char(1024) varz;
  Dcl OptStr2 char(1024) varz;
  Dcl myNull      Pointer;
  Dcl VM_Args     Like JavaVMInitArgs;
  Dcl myOptions   Like JavaVMOption;
  Dcl PLIXOPT     Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );

  Display('From PL/I  - Beginning execution of createJVM...');
  VM_Args.version = JNI_VERSION_1_6;

  myRC =  JNI_GetDefaultJavaVMInitArgs( addr(VM_Args) );
  OptStr1 = "-Djava.class.path=.:";
  OptStr2 = "-Djava.compiler=NONE";

  myOptions(1).theOptions = addr(OptStr1);
  myOptions(2).theOptions = addr(OptStr2);
  VM_Args.nOptions = 2;
  VM_Args.JavaVMOption = addr(myOptions);

  /* Create the Java VM                                        */
  myrc = JNI_CreateJavaVM(
             addr(jvm_ptr),
             addr(JNIEnv),
             addr(VM_Args) );

  /* Get the Java Class for the javaPart class                 */
  myClass = FindClass(JNIEnv, "javaPart");
  /* Get static method ID                                      */
  myMethodID = GetStaticMethodID(JNIEnv, myClass,"main",
               "([Ljava/lang/String;)V" );
  /* Create a Java String Object from the PL/I string.         */
  myJString = NewStringUTF(JNIenv, myPLIStr);
  myJObjArray = NewObjectArray(JNIEnv, 1,
                FindClass(JNIEnv,"java/lang/String"), myJString);
  Display('From PL/I  - Calling Java method in new JVM from PL/I...');
  Display('  ');
  myNull = CallStaticVoidMethod(JNIEnv, myClass,
             myMethodID, myJObjArray );
  /* destroy the Java VM                                       */
  Display('  ');
  Display('From PL/I  - Destroying the new JVM from PL/I...');
  myRC = DestroyJavaVM( JavaVM );
end;
```

Figure 93. PL/I sample program #4 - Calling the Java invocation API

Step 4: Compiling and linking the PL/I program

Procedure

1. Compile the PL/I sample program with the following command:

   ```
   pli -c createJVM.pli
   ```

2. Link the resulting PL/I object deck into a shared library with this command:

   ```
   c89 -o createJVM createJVM.o $JAVA_HOME/bin/classic/libjvm.x
   ```

   Notice the reference to the *$JAVA_HOME* environment variable. This variable should point to the directory where your Java 1.4 product is installed. For example, to set up this variable in your environment, you can use the following command:

   ```
   export JAVA_HOME="/usr/lpp/java/J6.0"
   ```

   In this case, the Java 1.4 product is assumed to be installed in `/usr/lpp/java/J6.0`.

Step 5: Running the sample program

Procedure

Run the Java-PL/I sample program with this command:

```
createJVM
```

The output of the sample program is as follows:

```
From PL/I  - Beginning execution of createJVM...
From PL/I  - Calling Java method in new JVM from PL/I...
From Java  - Hello World... ... a PLI string in a Java Virtual Machine!
From PL/I  - Destroying the new JVM from PL/I...
```

Attaching programs to an existing Java VM

You can use the Java invocation API to attach your program to an existing Java VM that was not created by your program.

About this task

When your PL/I application is running within an IMS JMP (Java messaging processing) region, IMS already created a Java VM. You can take the following steps to attach your program to the Java VM. Then you can call any functions that you need through the Java native interface.

Procedure

1. Locate the Java VM instance to attach your program to by using the **JNI_GetCreatedJavaVMs** API. In this IMS environment, only one Java VM is created, so you ask for only one Java VM pointer to be returned. You can code the call as follows:

   ```
   rc = JNI_GetCreatedJavaVMs( jvm_ptr, 1, nVMs );
   ```

   jvm_ptr
   : Is a pointer to a Java VM and is declared in the `IBMZJNI` include file. Java returns the address of the Java VM in this variable when the call returns successfully.

   nVMs
   : Is a fixed bin(31) variable that Java updates with the number of Java VM addresses when it returns. *nVMs* is 1 if the call is successful.

2. Acquire JNI environment pointer for the Java VM. You can use the **JGetEnv** function in the **JNIInvokeInterface_ structure** that is declared in the IBMZJNI include file:

```
rc = JGetEnv( jvm_ptr, JNIEnv, JNI_VERSION_1_6 );
```

jvm_ptr
Is the pointer to the Java VM instance.

JNIEnv
Is a pointer that Java sets to the JNI environment pointer for the Java VM instance.

JNI_VERSION_1_6
Is a constant holding the value of an interface version number, which is also declared in the IBMZJNI include file.

Results

Now that you have the JNI environment pointer for the Java VM, you can make calls to any of the JNI functions through the Java native interface.

Determining equivalent Java and PL/I data types

When you communicate with Java from PL/I, you need to match the data types between the two programming languages.

This table shows Java primitive types and their PL/I equivalents.

Table 36. Java primitive types and PL/I native equivalents

Java type	PL/I type	Size in Bits
Boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	21
double	jdouble	53
void	jvoid	n/a

Part 5. Specialized programming tasks

© Copyright IBM Corp. 1999, 2017

Chapter 19. Using the PLISAXA and PLISAXB XML parsers

The PLISAXx (x = A or B) built-in subroutines provide basic XML parsing capability, which allows programs to consume inbound XML documents, check them for well-formedness, and react to their contents.

These subroutines do not provide XML generation, which must instead be accomplished by the PL/I program logic or by using the XMLCHAR built-in function.

PLISAXA and PLISAXB have no special environmental requirements. They execute in all the principal runtime environments, including CICS, IMS, MQ Series, z/OS batch, and TSO.

PLISAXA and PLISAXB do have some important limits:

- They have no support for XML name spaces.
- They have no support for Unicode UTF-8 documents.
- They require that the entire XML document be passed to them (either in a buffer or a file) before they do any parsing of it.

The PLISAXC and PLISAXD built-in subroutines do not have these limits.

Related information:

Chapter 20, "Using the PLISAXC and PLISAXD XML parsers," on page 433
The PLISAXC and PLISAXD built-in subroutines provide basic XML parsing capability, which allows programs to consume inbound XML documents, check them for well-formedness, and react to their contents.

Overview

There are two major types of interfaces for XML parsing: event-based and tree-based.

For an event-based API, the parser reports events to the application through callbacks. Such events include the start of the document, the beginning of an element, and so on. The application provides handlers to deal with the events reported by the parser. The Simple API for XML (SAX) is an example of an industry-standard event-based API.

For a tree-based API such as the Document Object Model (DOM), the parser translates the XML into an internal tree-based representation. Interfaces are provided to navigate the tree.

IBM PL/I provides a SAX-like event-based interface for parsing XML documents. The parser invokes an application-supplied handler for parser events, passing references to the corresponding document fragments.

The parser has the following characteristics:

- It provides high-performance, but nonstandard interfaces.
- It supports XML files encoded in either Unicode UTF-16 or any of several single-byte code pages listed in "Coded character sets for XML documents" on page 409.

© Copyright IBM Corp. 1999, 2017

- The parser is nonvalidating, but does partially check well-formedness, and generates exception events if it discovers any.

XML documents have two levels of conformance: well-formedness and validity, both of which are defined in the XML standard, which you can find at http://www.w3c.org/XML/. Recapitulating these definitions, an XML document is well-formed if it complies with the basic XML grammar, and with a few specific rules, such as the requirement that the names on start and end element tags must match. A well-formed XML document is also valid if it has an associated document type declaration (DTD) and if it complies with the constraints expressed in the DTD.

For each parser event, you must provide a PL/I function that accepts the appropriate parameters and returns the appropriate return value - as in the example code shown in Figure 94 on page 405. Note in particular that the return value must be returned by value (BYVALUE). Also, these functions must all use the OPTLINK linkage. You can use the DEFAULT(LINKAGE(OPTLINK)) option to specify this linkage, or you can specify it on the individual PROCEDUREs and ENTRYs through the OPTIONS(LINKAGE(OPTLINK)) attribute.

The PLISAXA built-in subroutine

You can use the PLISAXA built-in subroutine to invoke the XML parser for an XML document that is in a buffer in your program.

e An event structure

p A pointer value or "token" that the parser will pass back to the event functions

x The address of the buffer containing the input XML

n The number of bytes of data in that buffer

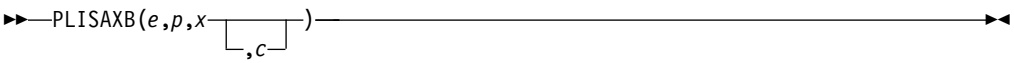
c A numeric expression specifying the purported codepage of that XML

Notes:

- If the XML is contained in a CHARACTER VARYING or WIDECHAR VARYING string, you must use the ADDRDATA built-in function to obtain the address of the first data byte.
- If the XML is contained in a WIDECHAR string, the value for the number of bytes is twice the value returned by the LENGTH built-in function.

The PLISAXB built-in subroutine

You can use the PLISAXB built-in subroutine to invoke the XML parser for an XML document in a file.



e An event structure

p A pointer value or "token" that the parser will pass back to the event functions

x A character string expression specifying the input file

c A numeric expression specifying the purported codepage of that XML

Under batch, the character string specifying the input file should have the form `file://dd:ddname`, where *ddname* is the name of the DD statement specifying the file.

Under z/OS UNIX, the character string specifying the input file should have the form `file://filename`, where *filename* is the name of a z/OS UNIX file.

Under both batch and z/OS UNIX, the character string specifying the input file should have no leading or trailing blanks.

The input XML file must be less than 2G in size. Moreover, because the parser will read the entire file into memory, the REGION for your program must be large enough for the parser to obtain a piece of storage that can contain all of the document.

The SAX event structure

The event structure is a structure consisting of 24 LIMITED ENTRY variables that point to functions that the parser will invoke for various "events".

All these ENTRYs must use the OPTLINK linkage.

The library routines that support the XML parsers recognize whether an event in any of the PLISAX event structures is set to null (through the NULLENTRY built-in function or through use of UNSPEC), and do not call it in that case. This allows you to limit your XML parsing code to only the events in which you are interested and to improve the performance of the overall parse at the same time.

```
xmlDocument =
   '<?xml version="1.0" standalone="yes"?>'
|| '<!--This document is just an example-->'
|| '<sandwich>'
|| '<bread type="baker&quot;s best"/>'
|| '<?spread please use real mayonnaise ?>'
|| '<meat>Ham &amp; turkey</meat>'
|| '<filling>Cheese, lettuce, tomato, etc.</filling>'
|| '<![CDATA[We should add a <relish> element in future!]]>'
|| '</sandwich>'
|| 'junk';
```

Figure 94. Sample XML document

In the order of their appearance in this structure, the parser might recognize the following events:

Note: The descriptions of each event refer to the example of an XML document in Figure 94. In these descriptions, the term *XML text* refers to the string based on the pointer and length passed to the event.

start_of_document

This event occurs once, at the beginning of parsing the document. The parser passes the address and length of the entire document, including any line-control

characters, such as LF (Line Feed) or NL (New Line). For the example shown in Figure 94 on page 405, the document is 305 characters in length.

version_information

This event occurs within the optional XML declaration for the version information. The parser passes the address and length of the text containing the version value (`1.0` for the example shown in Figure 94 on page 405).

encoding_declaration

This event occurs within the XML declaration for the optional encoding declaration. The parser passes the address and length of the text containing the encoding value.

standalone_declaration

This event occurs within the XML declaration for the optional standalone declaration. The parser passes the address and length of the text containing the standalone value (`yes` for the example shown in Figure 94 on page 405).

document_type_declaration

This event occurs when the parser finds a document type declaration. Document type declarations begin with the character sequence `<!DOCTYPE` and end with a `>` character, with some fairly complicated grammar rules describing the content in between. The parser passes the address and length of the text containing the entire declaration, including the opening and closing character sequences, and it is the only event where XML text includes the delimiters. The example shown in Figure 94 on page 405 does not have a document type declaration.

end_of_document

This event occurs once, when document parsing has completed.

start_of_element

This event occurs once for each element start tag or empty element tag. The parser passes the address and length of the text containing the element name. For the first start_of_element event during parsing of the example shown in Figure 94 on page 405, this is the string `sandwich`.

attribute_name

This event occurs for each attribute in an element start tag or empty element tag after the parser recognizes a valid name. The parser passes the address and length of the text containing the attribute name. The only attribute name in the example shown in Figure 94 on page 405 is `type`.

attribute_characters

This event occurs for each fragment of an attribute value. The parser passes the address and length of the text containing the fragment. An attribute value normally consists of a single string only, even if it is split across lines:

```
<element attribute="This attribute value is
split across two lines"/>
```

The attribute value might consist of multiple pieces, however. For instance, the value of the `type` attribute in the example shown in Figure 94 on page 405 consists of three fragments: the string `baker`, the single character `'`, and the string `s best`.

The parser passes these fragments as three separate events. It passes each string, `baker` and `s best`, as attribute_characters events, and the single character `'` as an attribute_predefined_reference event.

Related information:

"attribute_predefined_reference"

attribute_predefined_reference

This event occurs in attribute values for the five predefined entity references `&`, `'`, `>`, `<`, and `"`. The parser passes a CHAR(1) or WIDECHAR(1) value that contains one of `&`, `'`, `>`, `<`, or `"` respectively.

attribute_character_reference

This event occurs in attribute values for numeric character references (Unicode code points or "scalar values") of the form `&#dd;` or `&#xhh;`, where *d* represents decimal digits and *h* represents hexadecimal digits. The parser passes a FIXED BIN(31) value that contains the corresponding integer value.

end_of_element

This event occurs once for each element end tag or empty element tag when the parser recognizes the closing angle bracket of the tag. The parser passes the address and length of the text containing the element name.

start_of_CDATA_section

This event occurs at the start of a CDATA section. CDATA sections begin with the string `<![CDATA[` and end with the string `]]>`, and are used to "escape" blocks of text containing characters that would otherwise be recognized as XML markup. The parser passes the address and length of the text containing the opening characters `<![CDATA[`. The parser passes the content of a CDATA section between these delimiters as a single content-characters event. For the example shown in Figure 94 on page 405, the content-characters event is passed the text `We should add a <relish> element in future!`.

end_of_CDATA_section

This event occurs when the parser recognizes the end of a CDATA section. The parser passes the address and length of the text containing the closing character sequence, `]]>`.

content_characters

This event represents the "heart" of an XML document: the character data between element start and end tags. The parser passes the address and length of the text containing the data, which usually consists of a single string only, even if it is split across lines:

```
<element1>This character content is
split across two lines</element1>
```

If the content of an element includes any references or other elements, the complete content might comprise several segments. For instance, the content of the `meat` element in the example shown in Figure 94 on page 405 consists of the string `Ham ` , the character `&`, and the string ` turkey`. Notice the trailing and leading spaces, respectively, in these two string fragments. The parser passes these three content fragments as separate events. It passes the string content fragments, `Ham` and ` turkey`, as content_characters events, and the single `&` character as a

content_predefined_reference event. The parser also uses the content_characters event to pass the text of CDATA sections to the application.

content_predefined_reference

This event occurs in element content for the five pre-defined entity references `&`, `'`, `>`, `<`, and `"`. The parser passes a CHAR(1) or WIDECHAR(1) value that contains one of `&`, `'`, `>`, `<`, or `"`respectively.

content_character_reference

This event occurs in element content for numeric character references (Unicode code points or "scalar values") of the form `&#dd;` or `&#xhh;`, where *d* represents decimal digits and *h* represents hexadecimal digits. The parser passes a FIXED BIN(31) value that contains the corresponding integer value.

processing_instruction

Processing instructions (PIs) allow XML documents to contain special instructions for applications. This event occurs when the parser recognizes the name following the PI opening character sequence, `<?`. The event also covers the data following the processing instruction (PI) target, up to but not including the PI closing character sequence, `?>`. Trailing but not leading white space characters in the data are included. The parser passes the address and length of the text containing the target, `spread` in the example shown in Figure 94 on page 405, and passes the address and length of the text containing the data (`please use real mayonnaise` in the example).

comment

This event occurs for any comments in the XML document. The parser passes the address and length of the text between the opening comment delimiter `<!--` and the closing comment delimiter `-->`. In the example shown in Figure 94 on page 405, the text of the only comment is `This document is just an example`.

unknown_attribute_reference

This event occurs within attribute values for entity references other than the five predefined entity references, listed for the event attribute_predefined_character. The parser passes the address and length of the text containing the entity name.

unknown_content_reference

This event occurs within element content for entity references other than the five predefined entity references listed for the content_predefined_character event. The parser passes the address and length of the text containing the entity name.

start_of_prefix_mapping

This event is currently not generated.

end_of_prefix_mapping

This event is currently not generated.

exception

The parser generates this event when it detects an error in processing the XML document.

Parameters to the event functions

All of these functions must return a BYVALUE FIXED BIN(31) value that is a return code to the parser. For the parser to continue normally, this value should be zero.

All of these functions will be passed as the first argument a BYVALUE POINTER that is the token value passed originally as the second argument to the built-in function.

With the following exceptions, all of the functions will also be passed a BYVALUE POINTER and a BYVALUE FIXED BIN(31) that supply the address and length of the text element for the event. The following functions and events are different:

end_of_document
No argument other than the user token is passed.

attribute_predefined_reference
In addition to the user token, one additional argument is passed: a BYVALUE CHAR(1), or, for a UTF-16 document, a BYVALUE WIDECHAR(1) that holds the value of the predefined character.

content_predefined_reference
In addition to the user token, one additional argument is passed: a BYVALUE CHAR(1), or, for a UTF-16 document, a BYVALUE WIDECHAR(1) that holds the value of the predefined character.

attribute_character_reference
In addition to the user token, one additional argument is passed: a BYVALUE FIXED BIN(31) that holds the value of the numeric reference.

content_character_reference
In addition to the user token, one additional argument is passed: a BYVALUE FIXED BIN(31) that holds the value of the numeric reference.

processing_instruction
In addition to the user token, four additional arguments are passed:

1. A BYVALUE POINTER that is the address of the target text
2. A BYVALUE FIXED BIN(31) that is the length of the target text
3. A BYVALUE POINTER that is the address of the data text
4. A BYVALUE FIXED BIN(31) that is the length of the data text

exception
In addition to the user token, three additional arguments are passed:

1. A BYVALUE POINTER that is the address of the offending text
2. A BYVALUE FIXED BIN(31) that is the byte offset of the offending text within the document
3. A BYVALUE FIXED BIN(31) that is the value of the exception code

Coded character sets for XML documents

The PLISAX built-in subroutine supports only XML documents in WIDECHAR encoded in Unicode UTF-16, or in CHARACTER encoded in one of the explicitly supported single-byte character sets listed in this section.

The parser uses up to three sources of information about the encoding of your XML document, and signals an exception XML event if it discovers any conflicts between these sources:

1. The parser determines the basic encoding of a document by inspecting its initial characters.
2. If step 1 succeeds, the parser then looks for any encoding declaration.
3. Finally, it refers to the codepage value on the PLISAX built-in subroutine call. If this parameter was omitted, it defaults to the value provided by the CODEPAGE compiler option value that you specified explicitly or by default.

If the XML document begins with an XML declaration that includes an encoding declaration specifying one of the supported code pages listed in this section, the parser honors the encoding declaration if it does not conflict with either the basic document encoding or the encoding information from the PLISAX built-in subroutine. If the XML document does not have an XML declaration at all, or if the XML declaration omits the encoding declaration, the parser uses the encoding information from the PLISAX built-in subroutine to process the document, as long as it does not conflict with the basic document encoding.

Supported EBCDIC code pages

In the following table, the first number is for the Euro Country Extended Code Page (ECECP), and the second is for Country Extended Code Page (CECP).

CCSID	Description
01047	Latin 1 / Open Systems
01140, 00037	USA, Canada, etc.
01141, 00273	Austria, Germany
01142, 00277	Denmark, Norway
01143, 00278	Finland, Sweden
01144, 00280	Italy
01145, 00284	Spain, Latin America (Spanish)
01146, 00285	UK
01147, 00297	France
01148, 00500	International
01149, 00871	Iceland

Supported ASCII code pages

CCSID	Description
00813	ISO 8859-7 Greek / Latin
00819	ISO 8859-1 Latin 1 / Open Systems
00920	ISO 8859-9 Latin 5 (ECMA-128, Turkey TS-5881)

Specifying the code page

If your document does not include an encoding declaration in the XML declaration, or if it does not have an XML declaration at all, the parser uses the encoding information provided by the PLISAX built-in subroutine call together with the basic encoding of the document.

You can also specify the encoding information for the document in the XML declaration, with which most XML documents begin. The following example is an XML declaration that includes an encoding declaration:

```
<?xml version="1.0" encoding="ibm-1140"?>
```

If your XML document includes an encoding declaration, ensure that it is consistent with the encoding information provided by the PLISAX built-in subroutine and with the basic encoding of the document. If there is any conflict between the encoding declaration, the encoding information provided by the PLISAX built-in subroutine, and the basic encoding of the document, the parser signals an exception XML event.

You can specify the encoding declaration by using a number or an alias.

Specifying the encoding declaration using a number

You can specify the CCSID number (with or without any number of leading zeroes), prefixed by any of the following (in any mixture of uppercase or lowercase characters).

IBM_	CP	CCSID_
IBM-	CP_	CCSID-
	CP-	

Specifying the encoding declaration using an alias

You can use any of the following supported aliases (in any mixture of lowercase and uppercase characters).

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
813	ISO-8859-7, ISO_8859-7
819	ISO-8859-1, ISO_8859-1
920	ISO-8859-9, ISO_8859-9
1200	UTF-16

Exceptions

For most exceptions, the XML text contains the part of the document that was parsed up to including the point where the exception was detected. For encoding conflict exceptions, which are signaled before parsing begins, the length of the XML text is zero, or the XML text contains just the encoding declaration value from the document.

The example shown in Figure 94 on page 405 contains one item that causes an exception event, the superfluous `junk` following the `sandwich` element end tag.

There are two kinds of exceptions:

1. Exceptions that allow you to continue parsing optionally

Continuable exceptions have exception codes in the range 1 through 99, 100001 through 165535, or 200001 through 265535. The exception event in the example has an exception number of 1 and thus is continuable.

2. Fatal exceptions, which do not allow continuation

   Fatal exceptions have exception codes greater than 99 (but less than 100000).

Returning from the exception event function with a nonzero return code normally causes the parser to stop processing the document, and return control to the program that invoked the PLISAXA or PLISAXB built-in subroutine.

For continuable exceptions, returning from the exception event function with a zero return code requests the parser to continue processing the document, although further exceptions might subsequently occur. See "Continuable exception codes" on page 424 for details of the actions that the parser takes when you request continuation.

A special case applies to exceptions with exception numbers in the ranges 100001 through 165535 and 200001 through 265535. These ranges of exception codes indicate that the document's CCSID (determined by examining the beginning of the document, including any encoding declaration) is not identical to the CCSID value provided (explicitly or implicitly) by the PLISAXA or PLISAXB built-in subroutine, even if both CCSIDs are for the same basic encoding, EBCDIC or ASCII.

For these exceptions, the exception code passed to the exception event contains the document's CCSID, plus 100000 for EBCDIC CCSIDs, or 200000 for ASCII CCSIDs. For instance, if the exception code contains 101140, the document's CCSID is 01140. The CCSID value provided by the PLISAXA or PLISAXB built-in subroutine is set either explicitly as the last argument on the call or implicitly when the last argument is omitted and the value of the CODEPAGE compiler option is used.

Depending on the value of the return code after returning from the exception event function for these CCSID conflict exceptions, the parser takes one of three actions:

1. If the return code is zero, the parser proceeds using the CCSID provided by the built-in subroutine.
2. If the return code contains the document's CCSID (that is, the original exception code value minus 100000 or 200000), the parser proceeds using the document's CCSID. This is the only case where the parser continues after a nonzero value is returned from one of the parsing events.
3. Otherwise, the parser stops processing the document, and returns control to the PLISAXA or PLISAXB built-in subroutine, which will raise the ERROR condition.

Example

This example illustrates the use of the PLISAXA built-in subroutine.

The example uses the example XML document shown in Figure 94 on page 405.

```
saxtest: package exports(saxtest);

  define alias event
      limited entry( pointer, pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_end_of_document
      limited entry( pointer )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_predefined_ref
      limited entry( pointer, char(1) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) nodescriptor );

  define alias event_character_ref
      limited entry( pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_pi
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_exception
      limited entry( pointer, pointer, fixed bin(31),
                     fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );
```

Figure 95. PLISAXA coding example - type declarations

```
saxtest: proc options( main );

  dcl
    1 eventHandler static

      ,2 e01 type event
             init( start_of_document )
      ,2 e02 type event
             init( version_information )
      ,2 e03 type event
             init( encoding_declaration )
      ,2 e04 type event
             init( standalone_declaration )
      ,2 e05 type event
             init( document_type_declaration )
      ,2 e06 type event_end_of_document
             init( end_of_document )
      ,2 e07 type event
             init( start_of_element )
      ,2 e08 type event
             init( attribute_name )
      ,2 e09 type event
             init( attribute_characters )
      ,2 e10 type event_predefined_ref
             init( attribute_predefined_reference )
      ,2 e11 type event_character_ref
             init( attribute_character_reference )
      ,2 e12 type event
             init( end_of_element )
      ,2 e13 type event
             init( start_of_CDATA )
      ,2 e14 type event
             init( end_of_CDATA )
      ,2 e15 type event
             init( content_characters )
      ,2 e16 type event_predefined_ref
             init( content_predefined_reference )
      ,2 e17 type event_character_ref
             init( content_character_reference )
      ,2 e18 type event_pi
             init( processing_instruction )
      ,2 e19 type event
             init( comment )
      ,2 e20 type event
             init( unknown_attribute_reference )
      ,2 e21 type event
             init( unknown_content_reference )
      ,2 e22 type event
             init( start_of_prefix_mapping )
      ,2 e23 type event
             init( end_of_prefix_mapping )
      ,2 e24 type event_exception
             init( exception )
      ;
```

Figure 96. PLISAXA coding example - event structure

```
   dcl token       char(8);

   dcl xmlDocument char(4000) var;

   xmlDocument =
      '<?xml version="1.0" standalone="yes"?>'
   || '<!--This document is just an example-->'
   || '<sandwich>'
   || '<bread type="baker&quot;s best"/>'
   || '<?spread please use real mayonnaise ?>'
   || '<meat>Ham &amp; turkey</meat>'
   || '<filling>Cheese, lettuce, tomato, etc.</filling>'
   || '<![CDATA[We should add a <relish> element in future!]]>'.
   || '</sandwich>'
   || 'junk';


   call plisaxa( eventHandler,
                 addr(token),
                 addrdata(xmlDocument),
                 length(xmlDocument) );

end;
```

Figure 97. PLISAXA coding example - main routine

```
dcl chars char(32000) based;

start_of_document:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' length=' || tokenlength );

  return(0);
end;

version_information:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

encoding_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 98. PLISAXA coding example - event routines

```
  standalone_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  document_type_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  end_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;
```

PLISAXA coding example - event routines (continued)

```
  start_of_element:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  attribute_name:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  attribute_characters:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXA coding example - event routines (continued)

```
  attribute_predefined_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) nodescriptor );

    dcl userToken      pointer;
    dcl reference   char(1);

    put skip list( lowercase( procname() )
     || ' ' || hex(reference ) );

    return(0);
  end;

  attribute_character_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl reference      fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || hex(reference ) );

    return(0);
  end;

  end_of_element:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXA coding example - event routines (continued)

```
  start_of_CDATA:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  end_of_CDATA:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  content_characters:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXA coding example - event routines (continued)

```
  content_predefined_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) nodescriptor );

    dcl userToken      pointer;
    dcl reference   char(1);

    put skip list( lowercase( procname() )
     || ' ' || hex(reference ) );

    return(0);
  end;

  content_character_reference:
    proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl reference      fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || hex(reference ) );

    return(0);
  end;

  processing_instruction:
    proc( userToken, piTarget, piTargetLength,
                             piData, piDataLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl piTarget       pointer;
    dcl piTargetLength fixed bin(31);
    dcl piData         pointer;
    dcl piDataLength   fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

    return(0);
  end;
```

PLISAXA coding example - event routines (continued)

```
  comment:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  unknown_attribute_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  unknown_content_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXA coding example - event routines (continued)

```
  start_of_prefix_mapping:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  end_of_prefix_mapping:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  exception:
    proc( userToken, xmlToken, currentOffset, errorID )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl currentOffset  fixed bin(31);
    dcl errorID        fixed bin(31);

    put skip list( lowercase( procname() )
     || ' errorid =' || errorid );

    return(0);
  end;
end;
```

PLISAXA coding example - event routines (continued)

The preceding program produces the following output:

```
start_of_dcoument length=           305
version_information <1.0>
standalone_declaration <yes>
comment <This document is just an example>
start_of_element <sandwich>
start_of_element <bread>
attribute_name <type>
attribute_characters <baker>
attribute_predefined_reference 7D
attribute_characters <s best>
end_of_element <bread>
processing_instruction <spread>
start_of_element <meat>
content_characters <Ham >
content_predefined_reference 50
content_characters < turkey>
end_of_element <meat>
start_of_element <filling>
content_characters <Cheese, lettuce, tomato, etc.>
end_of_element <filling>
start_of_cdata <<![CDATA[>
content_characters <We should add a <relish> element in future!>
end_of_cdata <]]>
end_of_element <sandwich>
exception errorid =             1
content_characters <j>
exception errorid =             1
content_characters <u>
exception errorid =             1
content_characters <n>
exception errorid =             1
content_characters <k>
end_of_document
```

Figure 99. PLISAXA coding example - program output

Continuable exception codes

This topic describes the exception codes and the action that the parser takes when you request it to continue after the exception.

In the following table, each value of the exception code parameter passed to the exception event is listed in the **Number** column. For each exception code, a description is provided along with the actions that the parser takes when you request it to continue after the exception. In these descriptions, the term *XML text* refers to the string based on the pointer and length passed to the event.

Table 37. Continuable exceptions

Number	Description	Parser action on continuation
1	The parser found an invalid character while scanning white spaces outside element content.	The parser generates a content_characters event with the XML text containing the (single) invalid character. Parsing continues at the character after the invalid character.

Table 37. Continuable exceptions (continued)

Number	Description	Parser action on continuation
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.	The parser generates a content_characters event with the XML text containing the 2- or 3-character invalid initial character sequence. Parsing continues at the character after the invalid sequence.
3	The parser found a duplicate attribute name.	The parser generates an attribute_name event with the XML text containing the duplicate attribute name.
4	The parser found the markup character < in an attribute value.	Before generating the exception event, the parser generates an attribute_characters event for any part of the attribute value preceding the < character. After the exception event, the parser generates an attribute_characters event with the XML text containing <. Parsing then continues at the character after the <.
5	The start and end tag names of an element did not match.	The parser generates an end_of_element event with the XML text containing the mismatched end name.
6	The parser found an invalid character in element content.	The parser includes the invalid character in the XML text for the subsequent content_characters event.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.	Before generating the exception event, the parser generates a content_characters event for any part of the content preceding the < markup character. After the exception event, the parser generates a content_characters event with the XML text containing 2 characters: the < followed by the invalid character. Parsing continues at the character after the invalid character.
8	The parser found in element content the CDATA closing character sequence]]> without the matching opening character sequence <![CDATA[.	Before generating the exception event, the parser generates a content_characters event for any part of the content preceding the]]> character sequence. After the exception event, the parser generates a content_characters event with the XML text containing the 3-character sequence]]>. Parsing continues at the character after this sequence.
9	The parser found an invalid character in a comment.	The parser includes the invalid character in the XML text for the subsequent comment event.
10	The parser found in a comment the character sequence -- not followed by >.	The parser assumes that the -- character sequence terminates the comment, and generates a comment event. Parsing continues at the character after the -- sequence.
11	The parser found an invalid character in a processing instruction data segment.	The parser includes the invalid character in the XML text for the subsequent processing_instruction event.
12	A processing instruction target name was xml in lowercase, uppercase, or mixed-case.	The parser generates a processing_instruction event with the XML text containing xml in the original case.

Table 37. Continuable exceptions (continued)

Number	Description	Parser action on continuation
13	The parser found an invalid digit in a hexadecimal character reference (of the form `&#xdddd;`).	The parser generates an attribute_characters or content_characters event with the XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.
14	The parser found an invalid digit in a decimal character reference (of the form `&#dddd;`).	The parser generates an attribute_characters or content_characters event with the XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.
15	The encoding declaration value in the XML declaration did not begin with lowercase or uppercase A through Z.	The parser generates the encoding event with the XML text containing the encoding declaration value as it was specified.
16	A character reference did not refer to a legal XML character.	The parser generates an attribute_character_reference event or a content_character_reference event with XML-NTEXT containing the single Unicode character specified by the character reference.
17	The parser found an invalid character in an entity reference name.	The parser includes the invalid character in the XML text for the subsequent unknown_attribute_reference or unknown_content_reference event.
18	The parser found an invalid character in an attribute value.	The parser includes the invalid character in the XML text for the subsequent attribute_characters event.
50	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
51	The document was encoded in EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
52	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an ASCII encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
53	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
54	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.

Table 37. Continuable exceptions (continued)

Number	Description	Parser action on continuation
55	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
56	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
57	The document was encoded in ASCII, and the document encoding declaration specified a supported ASCII encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
58	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported EBCDIC encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
59	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
60	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
61	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
100001 through 165535	The document was encoded in EBCDIC; the encoding specified by the CODEPAGE compiler option and the encoding specified by the document encoding declaration are both supported EBCDIC code pages, but are not the same. The exception code contains the CCSID for the encoding declaration plus 100000.	If you return zero from the exception event, the parser uses the encoding specified by the CODEPAGE compiler option. If you return the CCSID from the document encoding declaration (by subtracting 100000 from the exception code), the parser uses this encoding.
200001 through 265535	The document was encoded in ASCII; the encoding specified by the CODEPAGE compiler option and the encoding specifies by the document encoding declaration are both supported ASCII code pages, but are not the same. The exception code contains the CCSID for the encoding declaration plus 200000.	If you return zero from the exception event, the parser uses the encoding specified by the CODEPAGE compiler option. If you return the CCSID from the document encoding declaration (by subtracting 200000 from the exception code), the parser uses this encoding.

Terminating exception codes

This topic describes the terminating exception codes.

Table 38. Terminating exceptions

Number	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or an entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or an entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, _, or :.
125	The first character of the first attribute name of an element was not a letter, _, or :.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than = following an attribute name.
128	The parser found an invalid attribute value delimiter.

Table 38. Terminating exceptions (continued)

Number	Description
130	The first character of an attribute name was not a letter, _, or :.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a > following the /.
133	The first character of an element end tag name was not a letter, _, or :.
134	An element end tag name was not terminated by a >.
135	The first character of an element name was not a letter, _, or :.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, _, or :.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence ?>.
141	The parser found an invalid character following & in a character reference or an entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by a =.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.
148	encoding in the XML declaration was not followed by a =.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional standalone declaration in the XML declaration.
153	'standalone' in the XML declaration was not followed by a =.
154	The standalone declaration value in the XML declaration is either missing or improperly delimited.
155	The standalone declaration value was neither yes nor no only.
156	The standalone declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
157	The parser found an invalid character following the standalone declaration value closing delimiter in the XML declaration.
158	The XML declaration was not terminated by the proper character sequence ?>, or contained an invalid attribute.

Table 38. Terminating exceptions (continued)

Number	Description
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
300	The document was encoded in EBCDIC, but the CODEPAGE compiler option specified a supported ASCII code page.
301	The document was encoded in EBCDIC, but the CODEPAGE compiler option specified Unicode.
302	The document was encoded in EBCDIC, but the CODEPAGE compiler option specified an unsupported code page.
303	The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document encoding declaration was either empty or contained an unsupported alphabetic encoding alias.
304	The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document did not contain an encoding declaration.
305	The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document encoding declaration did not specify a supported EBCDIC encoding.
306	The document was encoded in ASCII, but the CODEPAGE compiler option specified a supported EBCDIC code page.
307	The document was encoded in ASCII, but the CODEPAGE compiler option specified Unicode.
308	The document was encoded in ASCII, but the CODEPAGE compiler option did not specify a supported EBCDIC code page, ASCII, or Unicode.
309	The CODEPAGE compiler option specified a supported ASCII code page, but the document was encoded in Unicode.
310	The CODEPAGE compiler option specified a supported EBCDIC code page, but the document was encoded in Unicode.
311	The CODEPAGE compiler option specified an unsupported code page, but the document was encoded in Unicode.
312	The document was encoded in ASCII, but both the encoding provided externally and the encoding specified by the document encoding declaration are unsupported.
313	The document was encoded in ASCII, but the CODEPAGE compiler option is unsupported and the document did not contain an encoding declaration.
314	The document was encoded in ASCII, but the CODEPAGE compiler option is unsupported and the document encoding declaration did not specify a supported ASCII encoding.
315	The document was encoded in UTF-16 Little Endian, which the parser does not support on this platform.
316	The document was encoded in UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document may be damaged.
318	The document was encoded in UTF-8, which the parser does not support.
319	The document was encoded in UTF-16 Big Endian, which the parser does not support on this platform.
501, 502, 503	An internal error occurred in PLISAX(A|B). Contact IBM support.
500	Memory allocation failed for the PLISAXA internal data structures. Increase the amount of storage available to the application program.

Table 38. Terminating exceptions (continued)

Number	Description
520	Memory allocation failed for the PLISAXB internal data structures. Increase the amount of storage available to the application program.
521	An internal error occurred in PLISAX(A|B). Contact IBM support.
523	PLISAXB encountered a file I/O error.
524	Memory allocation failed in PLISAXB while attempting to cache the XML document from the file system. Increase the amount of storage available to the application program.
525	An unsupported URI scheme was specified to PLISAXB.
526	The XML document provided to PLISAXB was less than the minimum of 4 characters or was too large.
527, 560	An internal error occurred in PLISAX(A|B). Contact IBM support.
561	No event handler was specified to either PLISAX(A|B).
562, 563, 580, 581	An internal error occurred in PLISAX(A|B). Contact IBM support.
600 through 99999	Internal error. Report the error to your service representative.

Chapter 20. Using the PLISAXC and PLISAXD XML parsers

The PLISAXC and PLISAXD built-in subroutines provide basic XML parsing capability, which allows programs to consume inbound XML documents, check them for well-formedness, and react to their contents.

The XML parser used by PLISAXC is non-validating, but does partially check for well-formedness errors, and generates exception events if it discovers any.

The PLISAXD built-in subroutine provides XML parsing with validation capability. It determines whether an inbound XML documentation conforms to a set of rules specified in an inbound XML schema.

The PLISAXC and PLISAXD subroutines do not provide XML generation, which must instead be accomplished by the PL/I program logic or by the XMLCHAR built-in function.

PLISAXC and PLISAXD have no special environmental requirements except that it is not supported in AMODE 24. It executes in all the principal runtime environments, including CICS, IMS, MQ Series, z/OS batch, and TSO.

Because the PLISAXC and PLISAXD built-in subroutines and the PLISAXA and PLISAXB built-in subroutines do have much similarity, some information in this section repeats information in Chapter 19, "Using the PLISAXA and PLISAXB XML parsers," on page 403.

Overview

There are two major types of interfaces for XML parsing: event-based and tree-based.

For an event-based API, the parser reports events to the application through callbacks. Such events include the start of the document, the beginning of an element, and so on. The application provides handlers to deal with the events reported by the parser. The Simple API for XML (SAX) is an example of an industry-standard event-based API.

For a tree-based API such as the Document Object Model (DOM), the parser translates the XML into an internal tree-based representation. Interfaces are provided to navigate the tree.

IBM PL/I compiler provides, by using PLISAXC or PLISAXD, a SAX-like event-based interface for parsing XML documents. The parser invokes an application-supplied handler for parser events, passing references to the corresponding document fragments.

The parser has the following characteristics:

- It provides high-performance, but nonstandard interfaces.
- It supports XML files encoded in either Unicode UTF-16, UTF-8 or any of several single-byte code pages listed in "Coded character sets for XML documents" on page 442.

© Copyright IBM Corp. 1999, 2017

XML documents have two levels of conformance: well-formedness and validity, both of which are defined in the XML standard, which you can find at http://www.w3c.org/XML/. Recapitulating these definitions, an XML document is well-formed if it complies with the basic XML grammar, and with a few specific rules, such as the requirement that the names on start and end element tags must match. A well-formed XML document is also valid if it has an associated document type declaration (DTD) and if it complies with the constraints expressed in the DTD.

For each parser event, you must provide a PL/I function that accepts the appropriate parameters and returns the appropriate return value - as in the example code shown in Figure 100 on page 435.

Notes:

- For these functions, the return value must be returned by value (BYVALUE).
- For these functions, the linkage used must be the OPTLINK linkage. You can use the DEFAULT(LINKAGE(OPTLINK)) option to specify this linkage, or you can specify it on the individual PROCEDUREs and ENTRYs by using the OPTIONS(LINKAGE(OPTLINK)) attribute.

The PLISAXC built-in subroutine

The PLISAXC built-in subroutine allows you to invoke the XML parser for an XML document that is in one or more buffers in your program.

e An event structure

p A pointer value or "token" that the parser will pass back to the event functions

x The address of the initial buffer containing the input XML

n The number of bytes of data in that buffer

c A numeric expression specifying the codepage of that XML

Notes:

- If the XML is contained in a CHARACTER VARYING or WIDECHAR VARYING string, you must use the ADDRDATA built-in function to obtain the address of the first data byte.
- If the XML is contained in a WIDECHAR string, the value for the number of bytes is twice the value returned by the LENGTH built-in function.

The PLISAXD built-in subroutine

The PLISAXD built-in subroutine allows you to invoke the XML parser with the validation capability. Both the XML document and the Optimized Schema Representation (OSR) file are in one or more buffers in your program.

e An event structure

p A pointer value or "token" that the parser passes back to the event functions

x The address of the initial buffer containing the input XML

n The number of bytes of data in that buffer

o The address of the buffer containing the input OSR

c A numeric expression specifying the codepage of the XML document

Note:

- If the XML is contained in a CHARACTER VARYING or WIDECHAR VARYING string, you must use the ADDRDATA built-in function to obtain the address of the first data byte.
- If the XML is contained in a WIDECHAR string, the value for the number of bytes is twice the value returned by the LENGTH built-in function.
- An OSR is the preprocessed version of a schema. For more information about OSR, see *XML System Services User's Guide and Reference*.

The SAX event structure

The event structure is a structure consisting of 19 LIMITED ENTRY variables, which point to functions that the parser invokes for various "events".

All of these ENTRYs must use the OPTLINK linkage.

The library routines that support the XML parsers recognize if an event in any of the PLISAX event structures is set to null (through the NULLENTRY built-in function or through use of UNSPEC) and do not call it in that case. This allows you to limit your XML parsing code to only the events in which you are interested and to improve the performance of the overall parse at the same time.

All of these ENTRYs have a first (and sometimes the only) parameter: the user token passed by the program to PLISAXC and PLISAXD.

In this section, the descriptions of these 19 events refer to the example of an XML document in Figure 100. In these descriptions, the term *XML text* refers to the string based on the pointer and length passed to the event.

```
xmlDocument =
   '<?xml version="1.0" standalone="yes"?>'
|| '<!--This document is just an example-->'
|| '<sandwich>'
|| '<bread type="baker&quot;s best"/>'
|| '<?spread please use real mayonnaise ?>'
|| '<meat>Ham &amp; turkey</meat>'
|| '<filling>Cheese, lettuce, tomato, etc.</filling>'
|| '<![CDATA[We should add a <relish> element in future!]]>'
|| '</sandwich>';
```

Figure 100. Sample XML document

Depending on the contents of the XML documents, the parser might recognize the following events:

start_of_document

This event occurs once, at the beginning of parsing the document. The parser passes no parameters to this event (except the user token).

version_information

This event occurs within the optional XML declaration for the version information. The parser passes the address and length of the text containing the version value (`1.0` for the example shown in Figure 100 on page 435).

encoding_declaration

This event occurs within the XML declaration for the optional encoding declaration. The parser passes the address and length of the text containing the encoding value.

standalone_declaration

This event occurs within the XML declaration for the optional standalone declaration. The parser passes the address and length of the text containing the standalone value (`yes` for the example shown in Figure 100 on page 435).

document_type_declaration

This event occurs when the parser finds a document type declaration. Document type declarations begin with the character sequence `<!DOCTYPE` and end with a `>` character, with some fairly complicated grammar rules describing the content in between. The parser passes the address and length of the text containing the entire declaration, including the opening and closing character sequences, and it is the only event where XML text includes the delimiters. The example shown in Figure 100 on page 435 does not have a document type declaration.

end_of_document

This event occurs once, when document parsing has completed. The parser passes no parameters to this event (except the user token).

start_of_element

This event occurs once for each element start tag or empty element tag. The parser passes the address and length of the text containing the element name as well as any applicable namespace information. For the first start_of_element event during parsing of the example shown in Figure 100 on page 435, this is the string `sandwich`.

attribute_name

This event occurs for each attribute in an element start tag or empty element tag after the parser recognizes a valid name. The parser passes the address and length of the text containing the attribute name as well as any applicable namespace information. The only attribute name in the example shown in Figure 100 on page 435 is `type`.

attribute_characters

This event occurs for each attribute value. The parser passes the address and length of the text containing the fragment. An attribute value normally consists of a single string only, even if it is split across lines:

```
<element attribute="This attribute value is
split across two lines"/>
```

The parser also passes a flag byte, which indicates whether the next event provides additional characters that form part of the content. This can be true when there is a lot of data between the start and end tags.

end_of_element

This event occurs once for each element end tag or empty element tag when the parser recognizes the closing angle bracket of the tag. The parser passes the address and length of the text containing the element name as well as any applicable namespace information.

start_of_CDATA_section

This event occurs at the start of a CDATA section. CDATA sections begin with the string `<![CDATA[` and end with the string `]]>`, and are used to "escape" blocks of text containing characters that would otherwise be recognized as XML markup. The parser passes no parameters to this event (except the user token). After this event, the parser passes the content of the CDATA section between these delimiters as one or more content-characters events. For the example shown in Figure 100 on page 435, the content-characters event is passed the text `We should add a <relish> element in future!`.

end_of_CDATA_section

This event occurs when the parser recognizes the end of a CDATA section. The parser passes no parameters to this event (except the user token).

content_characters

This event represents the "heart" of an XML document: the character data between element start and end tags. The parser passes the address and length of the text containing the data, which usually consists of a single string only, even if it is split across lines:

```
<element1>This character content is
split across two lines</element1>
```

The parser also passes a flag byte, which indicates if the next event provides additional characters that form part of the content. This can be true when there is a lot of data between the start and end tags.

The parser also uses the content_characters event to pass the text of CDATA sections to the application.

processing_instruction

Processing instructions (PIs) allow XML documents to contain special instructions for applications. This event occurs when the parser recognizes the name following the PI opening character sequence `<?`. The event also covers the data following the processing instruction (PI) target, up to but not including the PI closing character sequence `?>`. Trailing but not leading white space characters in the data are included. The parser passes the address and length of the text containing the target, `spread` in the example shown in Figure 100 on page 435, and passes the address and length of the text containing the data (`please use real mayonnaise` in the example).

The parser also passes a flag byte, which indicates whether the next event provides additional characters that form part of the content. This can be true when there is a lot of data between the start and end tags.

comment

This event occurs for any comments in the XML document. The parser passes the address and length of the text between the opening comment delimiter `<!--` and the closing comment delimiter `-->`. In the example shown in Figure 100 on page 435, the text of the only comment is `This document is just an example`.

The parser also passes a flag byte, which indicates whether the next event provides additional characters that form part of the content. This can be true when there is a lot of data between the start and end tags.

namespace_declare

This event occurs for any namespace declarations in the XML document. The parser passes the address and length of the namespace prefix (if any) as well as the address and length of the namespace uri. If there is no namespace prefix, the passed length will be zero and the value of the address should not be used. There is no corresponding event in the PLIXSAXA and PLISAXB built-in subroutines.

end_of_input

This event occurs whenever the parser reaches the end of the current input buffer. The parser passes (along with the BYVALUE user token) two BYADDR parameters: the address and length of the next buffer for it to process. Note that this and the content character events are the only events that have any BYADDR parameters, but this is the only event that has parameters that the called event should change. There is no corresponding event in the PLIXSAXA and PLISAXB built-in subroutines, and it is this event that allows PLISAXC and PLISAXD to parse an XML document of arbitrary size.

unresolved_reference

This event occurs for any unresolved references in the XML document. The parser passes the address and length of the unresolved reference.

exception

The parser generates this event when it detects an error in processing the XML document.

Parameters to the event functions

All of these functions must return a BYVALUE FIXED BIN(31) value that is a return code to the parser. If any value other than zero is returned, the parser will terminate.

All of these functions are passed a BYVALUE POINTER as the first argument. This pointer is the token value that is passed originally as the second argument to the built-in function.

With the following exceptions, all of the functions will also be passed a BYVALUE POINTER and a BYVALUE FIXED BIN(31) that supply the address and length of the text element for the event. The following functions and events are different:

start_of_document
No argument other than the user token is passed.

end_of_document
No argument other than the user token is passed.

start_of_CDATA
No argument other than the user token is passed.

end_of_CDATA
No argument other than the user token is passed.

start_of_element
In addition to the usual three parameters, four additional arguments are passed:

1. A BYVALUE POINTER that is the address of the namespace prefix
2. A BYVALUE FIXED BIN(31) that is the length of the namespace prefix
3. A BYVALUE POINTER that is the address of the namespace uri
4. A BYVALUE FIXED BIN(31) that is the length of the namespace uri

end_of_element
In addition to the usual three parameters, four additional arguments are passed:

1. A BYVALUE POINTER that is the address of the namespace prefix
2. A BYVALUE FIXED BIN(31) that is the length of the namespace prefix
3. A BYVALUE POINTER that is the address of the namespace uri
4. A BYVALUE FIXED BIN(31) that is the length of the namespace uri

attribute_name
In addition to the usual three parameters, four additional arguments are passed:

1. A BYVALUE POINTER that is the address of the namespace prefix
2. A BYVALUE FIXED BIN(31) that is the length of the namespace prefix
3. A BYVALUE POINTER that is the address of the namespace uri
4. A BYVALUE FIXED BIN(31) that is the length of the namespace uri

attribute_characters
In addition to the usual three parameters, one additional argument is passed:

- a BYVALUE ALIGNED BIT(8) flag byte that indicates the following information:
  - Whether more content characters will be presented in the next event - this is true if the first bit is on, that is, this is true if this field anded with '80'BX is nonnull
  - Whether there are no characters that need to be escaped if converted back to XML - this is true if the second bit is on, that is, this is true if this field anded with '40'BX is nonnull

Note that this entry must also be declared with OPTIONS(NODESCRIPTOR).

namespace_declare
In addition to the user token, four additional arguments passed:

1. A BYVALUE POINTER that is the address of the namespace prefix
2. A BYVALUE FIXED BIN(31) that is the length of the namespace prefix
3. A BYVALUE POINTER that is the address of the namespace uri
4. A BYVALUE FIXED BIN(31) that is the length of the namespace uri

content_characters

In addition to the usual three parameters, one additional argument is passed:

- A BYVALUE ALIGNED BIT(8) flag byte that indicates the following information:
  - Whether more content characters will be presented in the next event - this is true if the first bit is on, that is, this is true if this field anded with '80'BX is nonnull
  - Whether there are no characters that need to be escaped if converted back to XML - this is true if the second bit is on, that is, this is true if this field anded with '40'BX is nonnull

Note that this entry must also be declared with OPTIONS(NODESCRIPTOR).

end_of_input

In addition to the user token, two additional arguments are passed:

1. A BYADDR POINTER that is the address of the next input buffer
2. A BYADDR FIXED BIN(31) that is the length of the next input buffer

processing_instruction

In addition to the user token, five additional arguments are passed:

1. A BYVALUE POINTER that is the address of the target text
2. A BYVALUE FIXED BIN(31) that is the length of the target text
3. A BYVALUE POINTER that is the address of the data text
4. A BYVALUE FIXED BIN(31) that is the length of the data text
5. A BYVALUE ALIGNED BIT(8) flag byte that indicates the following information:
   - Whether more content characters will be presented in the next event - this is true if the first bit is on, that is, this is true if this field anded with '80'BX is nonnull
   - Whether there are no characters that need to be escaped if converted back to XML - this is true if the second bit is on, that is, this is true if this field anded with '40'BX is nonnull

   Note that this entry must also be declared with OPTIONS(NODESCRIPTOR).

comment

In addition to the usual three parameters, one additional argument is passed:

- A BYVALUE ALIGNED BIT(8) flag byte that indicates the following information:
  - Whether more content characters will be presented in the next event - this is true if the first bit is on, that is, this is true if this field anded with '80'BX is nonnull
  - Whether there are no characters that need to be escaped if converted back to XML - this is true if the second bit is on, that is, this is true if this field anded with '40'BX is nonnull

Note that this entry must also be declared with OPTIONS(NODESCRIPTOR).

exception

In addition to the user token, three additional arguments are passed:

1. A BYVALUE FIXED BIN(31) that is the byte offset of the offending text within the document
2. A BYVALUE FIXED BIN(31) that is the return code for the exception
3. A BYVALUE FIXED BIN(31) that is the reason code for the exception

Differences in the events

The following events are part of the PLISAXA and PLISAXB event structure, but are not in PLISAXC and PLISAXD:

- attribute_predefined_reference
- attribute_character_reference
- content_predefined_reference
- content_character_reference
- unknown_attribute_reference
- unknown_content_reference
- start_of_prefix_mapping
- end_of_prefix_mapping

The following events are not part of the PLISAXA and PLISAXB event structure, but are in PLISAXC and PLISAXD:

- namespace_declare
- unresolved_reference
- end_of_input

Some of the events that are common to PLISAXA and PLISAXB, PLISAXC and PLISAXD are passed different parameters (apart from the omnipresent user token):

start_of_document
is passed no parameters

start_of_element
is passed namespace data as well

end_of_element
is passed namespace data as well

attribute_name
is passed namespace data as well

attribute_characters
is passed a flag byte as well

start_of_cdata
is passed no parameters

end_of_cdata
is passed no parameters

content_characters
is passed a flag byte as well

processing_instruction
is passed a flag byte as well

comment
is passed a flag byte as well

exception
is passed a return and reason code instead of an error id

Coded character sets for XML documents

The PLISAXC and PLISAXD built-in subroutines support only XML documents in WIDECHAR encoded in Unicode UTF-16 or in CHARACTER encoded in either UTF-8 or one of the explicitly supported single-byte character sets listed in this section.

The parser uses up to three sources of information about the encoding of your XML document, and signals an exception XML event if it discovers any conflicts between these sources:

1. The parser determines the basic encoding of a document by inspecting its initial characters.
2. If step 1 succeeds, the parser then looks for any encoding declaration.
3. Finally, it refers to the codepage value on the PLISAXC or PLISAXD built-in subroutine call. If this parameter was omitted, it defaults to the value provided by the CODEPAGE compiler option value that you specified explicitly or by default.

If the XML document begins with an XML declaration that includes an encoding declaration specifying one of the supported code pages, the parser honors the encoding declaration if it does not conflict with either the basic document encoding or the encoding information from the PLISAXC or PLISAXD built-in subroutine. If the XML document does not have an XML declaration at all, or if the XML declaration omits the encoding declaration, the parser uses the encoding information from the PLISAXC or PLISAXD built-in subroutine to process the document, as long as it does not conflict with the basic document encoding.

Supported code pages

In the following table, the first number is for the Euro Country Extended Code Page (ECECP), and the second is for Country Extended Code Page (CECP).

CCSID	Description
01208	Unicode UTF-8
01047	Latin 1 / Open Systems
01140, 00037	USA, Canada, etc.
01141, 00273	Austria, Germany
01142, 00277	Denmark, Norway
01143, 00278	Finland, Sweden
01144, 00280	Italy
01145, 00284	Spain, Latin America (Spanish)
01146, 00285	UK
01147, 00297	France
01148, 00500	International
01149, 00871	Iceland

Specifying the code page

If your document does not include an encoding declaration in the XML declaration, or if it does not have an XML declaration at all, the parser uses the

encoding information provided by the PLISAXC or PLISAXD built-in subroutine call along with the basic encoding of the document.

You can also specify the encoding information for the document in the XML declaration, with which most XML documents begin. The following example is an XML declaration that includes an encoding declaration:

```
<?xml version="1.0" encoding="ibm-1140"?>
```

If your XML document includes an encoding declaration, ensure that it is consistent with the encoding information provided by the PLISAXC or PLISAXD built-in subroutine and with the basic encoding of the document. If there is any conflict between the encoding declaration, the encoding information provided by the PLISAXC or PLISAXD built-in subroutine, and the basic encoding of the document, the parser signals an exception XML event.

You can specify the encoding declaration by using a number or an alias.

Specifying the encoding declaration using a number

You can specify the CCSID number (with or without any number of leading zeroes), prefixed by any of the following (in any mixture of uppercase or lowercase characters):

IBM_	CP	CCSID_
IBM-	CP_	CCSID-
	CP-	

Specifying the encoding declaration using an alias

You can use any of the following supported aliases (in any mixture of lowercase and uppercase characters).

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
813	ISO-8859-7, ISO_8859-7
819	ISO-8859-1, ISO_8859-1
920	ISO-8859-9, ISO_8859-9
1200	UTF-16

Exceptions

If an exception event occurs, the reason and return codes passed to it are those from the XML System Services parser, and the documentation provided with that parser explains what these return and reason codes mean.

Parsing XML documents with validation

The PLISAXD built-in subroutine not only parses XML documents in the same manner as PLISAXC, but also determines whether an inbound XML document conforms to a set of rules specified in an inbound XML schema.

When you use the PLISAXD built-in subroutine, the inbound schema used for XML validation must be in a preprocessed format known as an Optimized Schema Representation (OSR).

The following topics contain the description of the XML schema and the way to build an OSR for an XML schema. For an example of using PLISAXD built-in subroutine to parse XML documents with validation, see "Example of using the PLISAXD built-in subroutine" on page 455.

XML schema

An XML schema is a mechanism, defined by the W3C, for describing and constraining the structure and content of XML documents.

Through its support for datatypes and namespaces, an XML schema has the potential to provide the standard structure for XML elements and attributes. Therefore, an XML schema, which is itself expressed in XML, can effectively define a class of XML documents of a given type, for example, stock item.

The following sample XML document describes an item for stock keeping purposes:

```
     '<?xml version="1.0" standalone="yes"?>'
  || '<!--Document for stock keeping example-->'
  || '<stockItem itemNumber="453-SR">'
  || '<itemName>Stainless steel rope thimbles</itemName>'
  || '<quantityOnHand>23</quantityOnHand>'
  || '<stockItem>';
```

The stock keeping example document is both well formed and valid according to the following schema called `stock.xsd`. (The numbers that precede each line are not part of the schema, but are used in the explanation after the schema.)

```
1. <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2.
3. <xsd:element name="stockItem" type="stockItemType"/>
4.
5. <xsd:complexType name="stockItemType">
6.  <xsd:sequence>
7.   <xsd:element name="itemName" type="xsd:string" minOccurs="0"/>
8.   <xsd:element name="quantityOnHand">
9.    <xsd:simpleType>
10.    <xsd:restriction base="xsd:nonNegativeInteger">
11.      <xsd:maxExclusive value="100"/>
12.     </xsd:restriction>
13.    </xsd:simpleType>
14.   </xsd:element>
15.  </xsd:sequence>
16.  <xsd:attribute name="itemNumber" type="SKU" use="required"/>
17. </xsd:complexType>
18.
19. <xsd:simpleType name="SKU">
20.  <xsd:restriction base="xsd:string">
21.   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
22.  </xsd:restriction>
23. </xsd:simpleType>
24.
25. </xsd:schema>
```

The schema declares (line 3) that the root element is `stockItem`, which has a mandatory `itemNumber` attribute (line 16) of type `SKU`, and includes a sequence (lines 6 - 15) of other elements:

- An optional `itemName` element of type string (line 7)

- A required quantityOnHand element that has a constrained range of 1 - 99 based on the type nonNegativeInteger (lines 8 - 14)

Type declarations can be inline and unnamed, as in lines 9 - 13, which includes the maxExclusive facet to specify the legal values for the quantityOnHand element.

For the itemNumber attribute, by contrast, the named type SKU is declared separately in lines 19 - 23, which includes a pattern facet that uses regular expression syntax to specify that the legal values for that type consist of (in order) 3 digits, a hyphen-minus, and then two uppercase letters.

Creating an OSR

To generate a schema in the OSR format from a text-form schema, use the z/OS UNIX command **xsdosrg**, which invokes the OSR generator provided by z/OS UNIX System Services.

For example, to convert the text-form schema in the stock.xsd file to a schema in preprocessed format in the stock.osr file, you can use the following z/OS UNIX command:

```
xsdosrg -v –o /u/HLQ/xml/stock.osr  /u/HLQ/xml/stock.xsd
```

/u/HLQ/xml/ is the directory where the stock.osr and stock.xsd files are located.

If you want to copy the generated OSR file into a PDS, use the z/OS UNIX **cp** command. To specify an MVS data set name, precede the name with double slashes (//). For example, to copy the HFS file stock.osr into a fixed block record format PDS called HLQ.XML.OSR with record length 80, you can use the following command:

```
cp –p /u/HLQ/xml/stock.osr "//'HLQ.XML.OSR(STOCK)'"
```

To omit the fully qualified name of the PDS, you can eliminate the single quotation mark in the cp statement such as follows:

```
cp –p /u/HLQ/xml/stock.osr "//XML.OSR(STOCK)"
```

For more information, see the *XML System Services User's Guide and Reference*.

Example with a simple document

This section contains two examples that illustrate the use of the PLISAXC and PLISAXD built-in subroutines.

Example of using the PLISAXC built-in subroutine

This example illustrates the use of the PLISAXC built-in subroutine.

The example uses the example XML document shown in Figure 100 on page 435. This example does not use namespaces, and all the input is passed when PLISAXC is first invoked (and as a result, the end_of_input event should not be invoked).

```
saxtest: package exports(saxtest);

  define alias event
      limited entry( pointer, pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_with_flag
      limited entry( pointer, pointer, fixed bin(31),
                     bit(8) aligned )
      returns( byvalue fixed bin(31) )
      options( nodescriptor byvalue linkage(optlink) );

  define alias event_with_namespace
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_without_data
      limited entry( pointer )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_pi
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_namespace_dcl
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_exception
      limited entry( pointer, fixed bin(31),
                     fixed bin(31),
                     fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_end_of_input
      limited entry( pointer,
                     pointer byaddr,
                     fixed bin(31) byaddr )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );
```

Figure 101. PLISAXC coding example - type declarations

```
saxtest: proc options( main );


  dcl
    1 eventHandler static

      ,2 e01 type event_without_data
             init( start_of_document )

      ,2 e02 type event
             init( version_information )

      ,2 e03 type event
             init( encoding_declaration )

      ,2 e04 type event
             init( standalone_declaration )

      ,2 e05 type event
             init( document_type_declaration )

      ,2 e06 type event_without_data
             init( end_of_document )

      ,2 e07 type event_with_namespace
             init( start_of_element )

      ,2 e08 type event_with_namespace
             init( attribute_name )

      ,2 e09 type event_with_flag
             init( attribute_characters )

      ,2 e10 type event_with_namespace
             init( end_of_element )

      ,2 e11 type event_without_data
             init( start_of_CDATA )

      ,2 e12 type event_without_data
             init( end_of_CDATA )

      ,2 e13 type event_with_flag
             init( content_characters )

      ,2 e14 type event_pi
             init( processing_instruction )

      ,2 e15 type event_with_flag
             init( comment )

      ,2 e16 type event_namespace_dcl
             init( namespace_declare )

      ,2 e17 type event_end_of_input
             init( end_of_input )

      ,2 e18 type event
             init( unresolved_reference )

      ,2 e19 type event_exception
             init( exception )
      ;
```

Figure 102. PLISAXC coding example - event structure

```
   dcl token       char(8);

   dcl xmlDocument char(4000) var;

   xmlDocument =
      '<?xml version="1.0" standalone="yes"?>'
   || '<!--This document is just an example-->'
   || '<sandwich>'
   || '<bread type="baker's best"/>'
   || '<?spread please use real mayonnaise ?>'
   || '<meat>Ham &amp; turkey</meat>'
   || '<filling>Cheese, lettuce, tomato, etc.</filling>'
   || '<![CDATA[We should add a <relish> element in future!]]>'.
   || '</sandwich>'
   || ' ';


   call plisaxc( eventHandler,
                 addr(token),
                 addrdata(xmlDocument),
                 length(xmlDocument) );

end;
```

Figure 103. PLISAXC coding example - main routine

```
dcl chars char(32000) based;

start_of_document:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' length=' || tokenlength );

  return(0);
end;

version_information:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

encoding_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

standalone_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 104. PLISAXC coding example - event routines

```
  document_type_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  namespace_declare:
    proc( userToken, nsPrefix, nsPrefixLength,
                     nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() ) );
    put skip list( 'prefix = '
     || ' <' || substr(nsPrefix->chars,1,nsPrefixlength ) || '>' );
    put skip list( 'Uri = '
     || ' <' || substr(nsUri->chars,1,nsUrilength ) || '>' );

    return(0);
  end;

  end_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  start_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  attribute_name:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  end_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  content_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  attribute_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  start_of_CDATA:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

  end_of_CDATA:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  processing_instruction:
    proc( userToken,
          piTarget, piTargetLength,
          piData, piDataLength,
          flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl piTarget       pointer;
    dcl piTargetLength fixed bin(31);
    dcl piData         pointer;
    dcl piDataLength   fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  comment:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  unresolved_reference:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;
```

PLISAXC coding example - event routines (continued)

```
  exception:
    proc( userToken, currentOffset, return_code, reason_code )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl currentOffset  fixed bin(31);
    dcl return_code    fixed bin(31);
    dcl reason_code    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' return_code =' || hex(return_code)
     || ', reason_code =' || hex(reason_code)
     || ', offset =' || currentOffset );

    return(0);
  end;

  end_of_input:
    proc( userToken, addr_xml, length_xml )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl addr_xml       byaddr pointer;
    dcl length_xml     byaddr fixed bin(31);

    return(1);
  end;
end;
```

PLISAXC coding example - event routines (continued)

The preceding program produces the following output:

```
start_of_document
version_information <1.0>
standalone_declaration <yes>
comment <This document is just an example>
start_of_element <sandwich>
prefix =  <>
Uri =  <>
start_of_element <bread>
prefix =  <>
Uri =  <>
attribute_name <type>
prefix =  <>
Uri =  <>
attribute_characters <baker's best>
end_of_element <bread>
prefix =  <>
Uri =  <>
processing_instruction <spread>
piData =  <please use real mayonnaise >
start_of_element <meat>
prefix =  <>
Uri =  <>
content_characters <Ham & turkey>
end_of_element <meat>
prefix =  <>
Uri =  <>
start_of_element <filling>
prefix =  <>
Uri =  <>
content_characters <Cheese, lettuce, tomato, etc.>
!!flags = 01000000
end_of_element <filling>
prefix =  <>
Uri =  <>
start_of_cdata
content_characters <We should add a <relish> element in future >
end_of_cdata
end_of_element <sandwich>
prefix =  <>
Uri =  <>
end_of_document
```

Figure 105. PLISAXC coding example - program output

Example of using the PLISAXD built-in subroutine

This example illustrates the use of the PLISAXD built-in subroutine.

The example uses the example XML document shown in Figure 100 on page 435 and the XML schema cited in the examples in "Example of using the PLISAXC built-in subroutine" on page 445.

This example includes the validation of 8 different XML files against the same `stock.osr` schema. In the following output, you can see which XML documents in the `saxdtest` program fail validation against the schema.

The PLISAXD built-in subroutine requires the XML schema file to be read into a buffer. The OSR file in the following example is in a PDS. The initial size of the OSR buffer is set to 4096. If you have a larger OSR file, you can increase the initial size of the OSR buffer accordingly.

If the inbound schema file were in an HFS file instead, you could use the following code to read the OSR file into the buffer:

```
dcl osrin file input stream environment(u);
dcl fileddint builtin;
dcl fileread builtin;

/* Read the HFS OSR file into buffer*/

open file(osrin);
osr_length = fileddint( osrin, 'filesize');
osr_ptr = allocate(osr_length);
rc = fileread(osrin,osr_ptr,osr_length);
```

To run a program by using an OSR in a PDS, you can specify the following DD statement in the JCL:

```
//OSRIN DD DSN=HLQ.XML.OSR(STOCK),DISP=SHR
```

If the associated ddname OSRIN is an HFS file, use the following JCL statement instead:

```
//OSRIN DD PATH="/u/HLQ/xml/stock.osr"
```

```
saxdtest: package exports(saxdtest);
  /*************************************************************/
  /* saxdtest: Test PL/I XML validation support                */
  /* expected output:                                          */
  /*                                                           */
  /*   SAXDTEST: PL/I XML Validation sample                    */
  /*   SAXDTEST: Document Successfully parsed                  */
  /*   SAXDTEST: Document Successfully parsed                  */
  /*   Invalid: missing attribute itemNumber.                  */
  /*   exception return_code =00000018, reason_code =8613      */
  /*   Invalid: unexpected attribute warehouse.                */
  /*   exception return_code =00000018, reason_code =8612      */
  /*   Invalid: illegal attribute value 123-Ab.                */
  /*   exception return_code =00000018, reason_code =8809      */
  /*   Invalid: missing element quantityOnHand.                */
  /*   exception return_code =00000018, reason_code =8611      */
  /*   Invalid: unexpected element comment.                    */
  /*   exception return_code =00000018, reason_code =8607      */
  /*   Invalid: out-of-range element value 100                 */
  /*   exception return_code =00000018, reason_code =8803      */
  /*                                                           */
  /*************************************************************/

  define alias event
      limited entry( pointer, pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_with_flag
      limited entry( pointer, pointer, fixed bin(31),
                     bit(8) aligned )
      returns( byvalue fixed bin(31) )
      options( nodescriptor byvalue linkage(optlink) );
```

Figure 106. PLISAXD coding example - event routines

```
    define alias event_with_namespace
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_without_data
        limited entry( pointer )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_pi
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_namespace_dcl
        limited entry( pointer, pointer, fixed bin(31),
                       pointer, fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_exception
        limited entry( pointer, fixed bin(31),
                       fixed bin(31),
                       fixed bin(31) )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );

    define alias event_end_of_input
        limited entry( pointer,
                       pointer byaddr,
                       fixed bin(31) byaddr )
        returns( byvalue fixed bin(31) )
        options( byvalue linkage(optlink) );
```

PLISAXD coding example - event routines (continued)

```
saxdtest: proc options( main );

  dcl
    1 eventHandler static

      ,2 e01 type event_without_data
             init( start_of_document )

      ,2 e02 type event
             init( version_information )

      ,2 e03 type event
             init( encoding_declaration )

      ,2 e04 type event
             init( standalone_declaration )

      ,2 e05 type event
             init( document_type_declaration )

      ,2 e06 type event_without_data
             init( end_of_document )

      ,2 e07 type event_with_namespace
             init( start_of_element )

      ,2 e08 type event_with_namespace
             init( attribute_name )

      ,2 e09 type event_with_flag
             init( attribute_characters )

      ,2 e10 type event_with_namespace
             init( end_of_element )

      ,2 e11 type event_without_data
             init( start_of_CDATA )

      ,2 e12 type event_without_data
             init( end_of_CDATA )

      ,2 e13 type event_with_flag
             init( content_characters )

      ,2 e14 type event_pi
             init( processing_instruction )

      ,2 e15 type event_with_flag
             init( comment )

      ,2 e16 type event_namespace_dcl
             init( namespace_declare )

      ,2 e17 type event_end_of_input
             init( end_of_input )

      ,2 e18 type event
             init( unresolved_reference )

      ,2 e19 type event_exception
             init( exception )
      ;
```

PLISAXD coding example - event routines (continued)

```
dcl token       char(45);

dcl rc              fixed bin(31);
dcl i               fixed bin(31);
dcl xml_document(8) char(300) var;
dcl xml_valid_msg(8) char(45) var;
dcl osr_ptr         pointer;
dcl record          char(80);
dcl osr_index       fixed bin;
dcl osr_buf_tail    fixed bin;
dcl temp_osr        pointer;
dcl buf_size        fixed bin(31) init(4096);
dcl rec_size        fixed bin(31);
dcl osr_length      fixed bin(31);
dcl buf_length      fixed bin(31);
dcl eof fixed       bin(31) init(0);
dcl osrin           file input;

on endfile(osrin) begin;
  eof = 1;
end;

/* read the entire PDS osr file into the buffer */

put skip list ('SAXDTEST: PL/I XML Validation sample ');

osr_length = buf_size;
osr_index = 0;
osr_buf_tail = 0;
rec_size = length(record);

osr_ptr = allocate(osr_length);

do while (eof = 0 );

  read file(osrin) into(record);
  osr_buf_tail += rec_size;
  if osr_buf_tail > buf_size then
    do;
      buf_length = osr_length;
      osr_length +=buf_size;
      temp_osr = allocate(osr_length);

      call plimove(temp_osr, osr_ptr, buf_length);
      call plifree(osr_ptr);
      osr_ptr = temp_osr;
      osr_buf_tail = rec_size;

      call plimove(osr_ptr+osr_index, addr(record), rec_size);
      osr_index += rec_size;
    end;
  else
    do;
      call plimove(osr_ptr+osr_index, addr(record), rec_size);
      osr_index  +=rec_size;
    end;

end;
```

PLISAXD coding example - event routines (continued)

```
    /* Valid XMLFILE */
    xml_document(1) = '<stockItem itemNumber="453-SR">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '<quantityOnHand>23</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(1) = 'Valid XMLFILE ';
    /* Valid: the ITEMNAME element can be omitted */
    xml_document(2) = '<stockItem  itemNumber="453-SR">'
    || '<quantityOnHand>23</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(2) = 'Valid: the ITEMNAME element can be omitted.';

    /* Invalid: missing attribute itemNumber */
    xml_document(3) = '<stockItem>'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '<quantityOnHand>23</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(3) = 'Invalid: missing attribute itemNumber.';

    /* Invalid: unexpected attribute warehouse */
    xml_document(4) = '<stockItem itemNumber="453-SR" warehouse="NY">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '<quantityOnHand>23</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg4) = 'Invalid: unexpected attribute warehouse.';

    /* Invalid: illegal attribute value 123-Ab */
    xml_document(5) = '<stockItem itemNumber="123-Ab">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '<quantityOnHand>23</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(5) = 'Invalid: illegal attribute value 123-Ab.';

    /* Invalid: missing element quantityOnHand */
    xml_document(6) = '<stockItem itemNumber="074-UN">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '</stockItem>';

    xml_valid_msg(6) = 'Invalid: missing element quantityOnHand.';

    /* Invalid: unexpected element comment */
    xml_document(7) = '<stockItem itemNumber="453-SR">'
    || '<itemName>Stainless steel rope thimbles</itemName>'
    || '<quantityOnHand>1</quantityOnHand>'
    || '<commnet>Nylon bristles</comment>'
    || '</stockItem>';

    xml_valid_msg(7) = 'Invalid: unexpected element comment.';

    /* Invalid: out-of-range element value 100 */
    xml_document(8) = '<stockItem itemNumber="123-AB">'
    || '<itemName>Paintbrush</itemName>'
    || '<quantityOnHand>100</quantityOnHand>'
    || '</stockItem>';

    xml_valid_msg(8) = 'Invalid: out-of-range element value 100';
```

PLISAXD coding example - event routines (continued)

```
    do i = 1 to hbound(xml_document);;
      token = xml_valid_msg(i);
      call plisaxd( eventHandler,
                    addr(token),
                    addrdata(xml_document(i)),
                    length(xml_document(i)),
                    osr_ptr,
                    37 );
    end;

    close file(osrin);
    call plifree(osr_ptr);

  end;

  start_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    return(0);
  end;

  version_information:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    return(0);
  end;

  encoding_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    return(0);
  end;

  standalone_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  document_type_declaration:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);


    return(0);
  end;

  end_of_document:
    proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( 'SAXDTEST: Document Successfully parsed ');

    return(0);
  end;

  start_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
  attribute_name:
    proc( userToken, xmlToken, tokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;

  attribute_characters:
    proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodescriptor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
     ||' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

    if flags = ''b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

  end_of_element:
    proc( userToken, xmlToken, TokenLength,
          nsPrefix, nsPrefixLength,
          nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

```
start_of_CDATA:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;


  return(0);
end;

end_of_CDATA:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;

  return(0);
end;

content_characters:
  proc( userToken, xmlToken, TokenLength, flags )
  returns( byvalue fixed bin(31) )
  options( nodescriptor, byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);
  dcl flags          bit(8) aligned;

  if flags = ''b then;
  else

  return(0);
end;

processing_instruction:
  proc( userToken,
        piTarget, piTargetLength,
        piData, piDataLength,
        flags )
  returns( byvalue fixed bin(31) )
  options( nodescriptor, byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl piTarget       pointer;
  dcl piTargetLength fixed bin(31);
  dcl piData         pointer;
  dcl piDataLength   fixed bin(31);
  dcl flags          bit(8) aligned;

  put skip list( lowercase( procname() )
   || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

  if flags = ''b then;
  else
    put skip list( '!!flags = ' || flags );

  return(0);
end;
```

PLISAXD coding example - event routines (continued)

```
comment:
  proc( userToken, xmlToken, TokenLength, flags )
  returns( byvalue fixed bin(31) )
  options( nodescriptor, byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);
  dcl flags          bit(8) aligned;

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  if flags = ''b then;
  else
    put skip list( '!!flags = ' || flags );    return(0);
end;

namespace_declare:
  proc( userToken, nsPrefix, nsPrefixLength,
                   nsUri, nsUriLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl nsPrefix       pointer;
  dcl nsPrefixLength fixed bin(31);
  dcl nsUri          pointer;
  dcl nsUriLength    fixed bin(31);

  return(0);
end;

unresolved_reference:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  return(0);
end;

exception:
  proc( userToken, currentOffset, return_code, reason_code )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl currentOffset  fixed bin(31);
  dcl return_code    fixed bin(31);
  dcl reason_code    fixed bin(31);
  dcl validmsg       char(45) based;

  put skip list( userToken -> validmsg);
  put skip list( lowercase( procname() )
   || ' return_code =' || hex(return_code)
   || ', reason_code =' || substr(hex(reason_code),5,4));

  return(0);
end;
```

PLISAXD coding example - event routines (continued)

```
  end_of_input:
    proc( userToken, addr_xml, length_xml )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl addr_xml       byaddr pointer;
    dcl length_xml     byaddr fixed bin(31);

    return(0);
  end;
```

PLISAXD coding example - event routines (continued)

The following output shows the result of the sample program. For those documents that are not valid, the PLISAXD built-in subroutine invokes the XML exception event with the return code and reason code listed. For a detailed description of each return code and reason code, see *XML System Services User's Guide and Reference*.

```
SAXDTEST: PL/I XML Validation sample
SAXDTEST: Document Successfully parsed
SAXDTEST: Document Successfully parsed
Invalid: missing attribute itemNumber.
exception return_code =00000018, reason_code =8613
Invalid: unexpected attribute warehouse.
exception return_code =00000018, reason_code =8612
Invalid: illegal attribute value 123-Ab.
exception return_code =00000018, reason_code =8809
Invalid: missing element quantityOnHand.
exception return_code =00000018, reason_code =8611
Invalid: unexpected element comment.
exception return_code =00000018, reason_code =8607
Invalid: out-of-range element value 100
exception return_code =00000018, reason_code =8803
```

Figure 107. Output from PLISAXD sample

Chapter 21. Using PLIDUMP

This section provides information about dump options and the syntax used to call PLIDUMP, and describes PL/I-specific information included in the dump that can help you debug your routine.

Note: PLIDUMP conforms to National Language Support standards.

Figure 108 shows an example of a PL/I routine calling PLIDUMP to produce a z/OS Language Environment dump. In this example, the main routine PLIDMP calls PLIDMPA, which then calls PLIDMPB. The call to PLIDUMP is made in routine PLIDMPB.

```
%PROCESS MAP SOURCE STG LIST OFFSET LC(101);
 PLIDMP: PROC  OPTIONS(MAIN) ;

   Declare   (H,I) Fixed bin(31) Auto;
   Declare   Names Char(17) Static init('Bob Teri Bo Jason');
   H = 5;  I = 9;
   Put skip list('PLIDMP Starting');
   Call PLIDMPA;

     PLIDMPA:  PROC;
       Declare  (a,b)  Fixed bin(31)   Auto;
       a = 1;  b = 3;
       Put skip list('PLIDMPA Starting');
       Call PLIDMPB;

         PLIDMPB:  PROC;
           Declare  1 Name  auto,
                      2  First   Char(12) Varying,
                      2  Last    Char(12) Varying;
           First = 'Teri';
           Last  = 'Gillispy';
           Put skip list('PLIDMPB Starting');
           Call PLIDUMP('TBFC','PLIDUMP called from procedure PLIDMPB');
           Put Data;
         End PLIDMPB;

     End PLIDMPA;

 End PLIDMP;
```

Figure 108. Example PL/I routine calling PLIDUMP

The syntax and options for PLIDUMP are as follows:

```
►►—PLIDUMP—(character-string-expression 1,character-string-expression 2)————►◄
```

character-string-expression 1
A dump options character string consisting of one or more of the following options:

A Requests information relevant to all tasks in a multitasking program.

B BLOCKS (PL/I hexadecimal dump).

© Copyright IBM Corp. 1999, 2017

C Continue. The routine continues after the dump.

E Exit from current task of a multitasking program. The program continues to run after the requested dump is completed.

F FILES.

H STORAGE.

This includes all Language Environment storage, and hence all the BASED and CONTROLLED storage acquired through ALLOCATE statements.

Note: A ddname of CEESNAP should be specified with the H option to produce a SNAP dump of a PL/I routine, but if this is omitted, Language Environment will issue a message but still produce a dump with much very useful information.

K BLOCKS (when running under CICS). The Transaction Work Area is included.

NB NOBLOCKS.

NF NOFILES.

NH NOSTORAGE.

NK NOBLOCKS (when running under CICS).

NT NOTRACEBACK.

O Only information relevant to the current task in a multitasking program.

S Stop. The enclave is terminated with a dump.

T TRACEBACK.

T, F, and C are the default options.

character-string-expression 2
A user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP usage notes

If you use PLIDUMP, the following considerations apply:

- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- A DD statement with the ddname PLIDUMP, PL1DUMP, or CEEDUMP can be used to define the data set for the dump.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 133 to prevent dump records from wrapping. If SYSOUT is used as the target in any one of these DDs, you must specify MSGFILE(SYSOUT,FBA,133,0) or MSGFILE(SYSOUT,VBA,137,0) to ensure that the lines are not wrapped.
- When you specify the H option in a call to PLIDUMP, the PL/I library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.

- Support for SNAP dumps using PLIDUMP is only provided under z/OS. SNAP dumps are not produced in a CICS environment.
  - If the SNAP is not successful, the CEE3DMP DUMP file displays the message:

    ```
    Snap was unsuccessful
    ```

  - If the SNAP is successful, CEE3DMP displays the message:

    ```
    Snap was successful; snap ID = nnn
    ```

    where *nnn* corresponds to the SNAP identifier described above. An unsuccessful SNAP does not increment the identifier.
- If you want the program unit name, program unit address, and program unit offset to be listed correctly in the dump traceback table, ensure that your PL/I program unit is compiled with a compile-time option other than TEST(NONE,NOSYM). For example, you can specify the option as TEST(NOSYM,NOHOOK,BLOCK).

If you want to ensure portability across system platforms, use PLIDUMP to generate a dump of your PL/I routine.

Locating variables in the PLIDUMP output

To find variables in the PLIDUMP output, you should compile your program with the MAP option. The MAP option will cause the compiler to add to the listing a table showing the offset within AUTOMATIC and STATIC storage of all level-1 variables that are AUTOMATIC or STATIC.

To find a variable that is an element in a structure, it is also useful if you compile your program with the AGGREGATE option. This option will cause the compiler to add to the listing a table showing the offsets of all the elements of all the structures in your program.

Locating AUTOMATIC variables

To find an AUTOMATIC variable in the dump, you should find its offset within automatic using the output from the MAP option (and if necessary the AGGREGATE option).

If PLIDUMP has been invoked with the B option, the dump output will contain a hex dump of the dynamic save area (DSA) for each block. This is the automatic storage for that block.

For example, consider the following simple program:

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31);
     5.0        dcl b fixed bin(31);
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBC');
    10.0          end;
    11.0
    12.0        a = 0;
    13.0        b = 29;
    14.0        b = 17 / a;
```

The result of the compiler MAP option for this program looks like this, except that there is actually one more column on the right and the columns are actually spaced much further apart:

```
* * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = automatic,  Location = 160 : 0xA0(r13),
 B          1-0:5      Class = automatic,  Location = 164 : 0xA4(r13),
```

So, A is located at hex A0 off of register 13 and B is located at hex A4 off of register 13, where register 13 points to the DSA.

Because in this program PLIDUMP is called with the B option, it will include a hexadecimal dump of automatic storage for each block in the current calling chain. This will look like the following (again with the right columns cutoff):

```
Dynamic save area (TEST): 0AD963C8
      +000000 0AD963C8  10000000 0AD96188 00000000 00000000
      +000020 0AD963E8  00000000 00000000 00000000 00000000
      +000040 0AD96408  00000000 00000000 00000000 0AD96518
      +000060 0AD96428  00000000 00000000 00000000 00000000
      +000080 0AD96448 - +00009F 0AD96467   same as above
      +0000A0 0AD96468  00000000 0000001D 00100000 00000000
      +0000C0 0AD96488  0B300000 0A700930 0AD963C8 00000000
      +0000E0 0AD964A8  00000000 00000000 00000000 00000000
      +000100 0AD964C8  0AA47810 0A70E6D0 0AD96540 0AD960F0
      +000120 0AD964E8  00000001 0A70F4F8 0AD96318 00000000
      +000140 0AD96508  00000000 00000000 00000000 00000000
```

Because A is at hex offset A0 and B is at hex offset A4 in AUTOMATIC, the dump shows that A and B have the (expected) hex values of 00000000 and 0000001D respectively.

Note that under the compiler options OPT(2) and OPT(3), some variables, particularly FIXED BIN and POINTER scalar variables, might never be allocated storage and thus could not be found in the dump output.

Locating STATIC variables

If you compiled your code with the RENT option, static variables are located in the Writeable Static Area (WSA) for the current load module.

The offset of a variable within the WSA can be found from the output of the MAP option, and the WSA is held in the Language Environment control block called the CAA. The value of the WSA is also listed in the Language Environment dump.

However, if you compiled your code with the NORENT option, EXTERNAL STATIC is found as usual (using the linker listing and the output of the compiler's MAP option). INTERNAL STATIC will be dumped as part of the Language Environment dump (if PLIDUMP was called with the B option).

Note that unlike the older PL/I compilers, the address of static is not dedicated to any one register.

For example, consider the program above with the variables changed to STATIC:

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31) static;
```

```
  5.0        dcl b fixed bin(31) static;
  6.0
  7.0        on error
  8.0          begin;
  9.0            call plidump('TFBC');
 10.0          end;
 11.0
 12.0        a = 0;
 13.0        b = 29;
 14.0        b = 17 / a;
```

When the program is compiled with the NORENT option, the result of the compiler MAP option for this program looks like this, except that there is actually one more column on the right and the columns are actually spaced much further apart:

```
* * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = static,  Location = 0 : 0x0 + CSECT ***TEST2
 B          1-0:5      Class = static,  Location = 4 : 0x4 + CSECT ***TEST2
```

So, A is located at hex offset 00 into the static CSECT for the compilation unit TEST while B is located at hex offset 04.

Because in this program PLIDUMP is called with the B option, it will include a hexadecimal dump of static storage for each compilation in the current calling chain. This will look like (again with the right columns cutoff):

```
Static for procedure TEST    Timestamp: 2004.08.12
      +000000 0FC00AA0  00000000 0000001D 0FC00DC8 0FC00AC0
      +000020 0FC00AC0  0FC00AA8 00444042 00A3AE01 0FC009C8
      +000040 0FC00AE0  6E3BFFE0 00000000 00000000 00000000
      +000060 0FC00B00  00000000 00000000 00000000 00000000
      +000000 0AD963C8  10000000 0AD96188 00000000 00000000
```

So, A at hex offset 00 has the (expected) hex value 00000000, and B at hex offset 04 has the (also expected) hex value 0000001D or the decimal value 29.

Locating CONTROLLED variables

CONTROLLED variables are essentially LIFO stacks, and each CONTROLLED variable has an "anchor" that points to the top of that stack. The key to locating a CONTROLLED variable is to locate this anchor, and its location depends on the compiler options.

In the rest of this discussion of CONTROLLED variables, the program source is the same program as in Figure 108 on page 467, but with the storage class changed to CONTROLLED:

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31) controlled;
     5.0        dcl b fixed bin(31) controlled;
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBHC');
    10.0          end;
    11.0
```

```
   12.0        allocate a, b;
   13.0        a = 0;
   14.0        b = 29;
   15.0        b = 17 / a;
```

Under NORENT WRITABLE

The result of the compiler MAP option looks like this, except that again there is actually one more column on the right and the columns are actually spaced much further apart:

```
* * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = static,  Location = 8 : 0x8 + CSECT ***TEST2
 B          1-0:5      Class = static,  Location = 12 : 0xC + CSECT ***TEST2
```

Note that these lines describe the location of the anchors for A and B (not the location of A and B themselves). So the anchor for A is located at hex 08 into the static CSECT for the compilation unit TEST, and the anchor for B is located at hex 0C.

If PLIDUMP is called with the B option, it will include a hexadecimal dump of static storage for each compilation in the current calling chain. This will look like (again with the right columns cutoff):

```
Static for procedure TEST    Timestamp: . . .

  +000000 0FC00A88  0FC00DB0 0FC00AA8 102B8A30 102B8A50
  +000020 0FC00AA8  0FC00A88 00444042 00A3AE01 0FC009B0
  +000040 0FC00AC8  6E3BFFE0 00000000 00000000 00000000
```

So the anchor for A is at 102B8A30 and the anchor for B is at 102B8A50. But because these are CONTROLLED variables, their storage was obtained through ALLOCATE statements, and hence these addresses point into heap storage. But If PLIDUMP is called with the H option, it will include a hexadecimal dump of heap storage. This will look like (again with the right columns cutoff):

```
Enclave Storage:
  Initial (User) Heap
    +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
      . . .
    +001A00 102B8A18  102B7018 00000020 0FC00A90 00000014
                      00000000 00000000 00000000 00000000
    +001A20 102B8A38  102B7018 00000020 0FC00A94 00000014
                      00000000 00000000 0000001D 00000000
```

Because the anchor for A was at 102B8A30, A has the hex value 00000000, and because the anchor for B was at 102B8A50, B has the (expected) hex value 0000001D.

Under NORENT NOWRITABLE(FWS)

The result of the compiler MAP option under these options looks like this, except that again there is actually one more column on the right and the columns are actually spaced much further apart:

```
 * * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A          1-0:4      Class = automatic,  Location = 236 : 0xEC(r13)
 B          1-0:5      Class = automatic,  Location = 240 : 0xF0(r13)
```

Note: Under these options, there is an extra level of indirection in locating CONTROLLED variables, and hence the lines above describe the locations of the addresses of the anchors for A and B. So the address of the anchor for A is located at hex EC into the automatic for the block TEST, and the anchor for B is located at hex F0.

Because PLIDUMP is called with the B option, it will include a hexadecimal dump of automatic storage for each block in the current calling chain. This will look like (again with the right columns cutoff):

```
Dynamic save area (TEST): 102973C8
 +000000 102973C8  10000000 10297188 00000000 8FC007DA
         ....
 +0000E0 102974A8  0FC00998 00000000 00000000 102B8A40
                   102B8A28 10297030 102977D0 8FDF3D7E
```

So the address of the anchor for A is 102B8A40 and the address of the anchor for B is 102B8A28.

Because PLIDUMP was also called with the H option, it will include a hexadecimal dump of heap storage. This will look like (again with the right columns cutoff):

```
Enclave Storage:
   Initial (User) Heap
     +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
       . . .
     +001A00 102B8A18  102B7018 00000018 00000000 0FC00A78
                       102B8A80 00000000 102B7018 00000018
     +001A20 102B8A38  102B8A20 0FC00A74 102B8A60 00000000
                       102B7018 00000020 102B8A40 00000014
     +001A40 102B8A58  00000000 00000000 00000000 00000000
                       102B7018 00000020 102B8A28 00000014
     +001A60 102B8A78  00000000 00000000 0000001D 00000000
                       00000000 00000000 00000000 00000000
```

Because the address of the anchor for B was at 102B8A28, the anchor for B is at 102B8A80, and B has, as expected, the hex value 0000001D or decimal 29.

Under NORENT NOWRITABLE(PRV)

The MAP listing when the program is compiled with these options would look like this:

```
* * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 ***TEST3     1-0:4  Class = ext def,  Location = CSECT ***TEST3
 ***TEST4     1-0:5  Class = ext def,  Location = CSECT ***TEST4
_PRV_OFFSETS  1-0:1  Class = static,   Location = 8 : 0x8 + CSECT ***TEST2
```

The key here is the last line in this output: _PRV_OFFSETS is a static table that holds the offset into the PRV table for each CONTROLLED variable. This static table is generated only if the MAP option is specified.

To interpret this table, the compiler will also produce, immediately after the block names table, another, usually small, listing, which for our program would look like this:

```
          PRV Offsets

   Number   Offset Name
        1        8 A
        1        C B
```

This table lists the hex offset within the runtime _PRV_OFFSETS table for each of the named CONTROLLED variables. The block number (in the first column) can be used to distinguish variables with the same name but declared in different blocks.

Because the _PRV_OFFSETS table is in static storage (at hex offset 8) and because PLIDUMP was called with the B option, it will appear in the dump output, which would look like this:

```
Static for procedure TEST    Timestamp: . . .
  +000000 10908EC8  02020240 00000005 6DD7D9E5 6DD6C6C6
                    00000000 00000004 D00000A0 00100000
  +000020 10908EE8  6E3BFFE0 00000000 00000000 00000000
                    00000000 90010000 00000000 00000000
```

So the offset of A in the PRV table is 0, and the offset of B in the PRV table is 4. Note also the eyecatcher "_PRV_OFF" that occupies the first 8 bytes of the _PRV_OFFSETs table.

The PRV table is always located at offset 4 within the CAA, which, because PLIDUMP was called with the H option, will be in the dump output. The CAA looks like this:

```
Control Blocks Associated with the Thread:
  CAA: 0A7107D0
   +000000 0A7107D0  00000800 0ADB7DE0 0AD97018 0ADB7018
                     00000000 00000000 00000000 00000000
```

So the address of the PRV table is 0ADB7DE0, and it will also be in the dump output amongst the HEAP storage:

```
 Enclave Storage:
    Initial (User) Heap
      +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
        . . .
      +000DC0 0ADB7DD8  00000000 00000000 0ADB8A38 0ADB8A58
                        0ADB7018 00000488 00000000 00000000
```

So the PRV table contains 0ADB8A38 0ADB8A58 and so on, and because, as derived from the _PRV_OFFSETS table, the offset of A into the PRV table is 0 and the offset of B is 4, these are also the addresses of A and B respectively.

These addresses will also appear in the HEAP storage in the dump:

```
  Enclave Storage:
    Initial (User) Heap
      +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
        . . .
      +001A00 0ADB8A18  00000000 00000000 0ADB7018 00000020
                        00000000 00000014 0A7107D4 00000000
      +001A20 0ADB8A38  00000000 00000000 0ADB7018 00000020
                        00000004 00000014 0A7107D4 00000000
      +001A40 0ADB8A58  0000001D 00000000 00000000 00000000
                        00000000 00000000 00000000 00000000
```

So because the address of A is 0ADB8A38, the hex value of A is, as expected, 00000000, and because the address of B is 0ADB8A58, the hex value of B is, also as expected, 0000001D.

Saved compilation data

During a compilation, the compiler saves various information about the compilation in the load module. This information can be very useful in debugging and in future migration efforts. This section describes the information saved.

Copyright

If you specify the COPYRIGHT compiler option, the compiler will save the COPYRIGHT string as a CHARACTER VARYING string placed immediately before the timestamp data in the object.

This string will be followed by as many blanks as necessary so that the string plus these blanks will occupy a multiple of 4 bytes.

Timestamp

The compiler saves in every load module a *timestamp*, which is a 20-byte character string of the form *YYYYMMDDHHMISSVNRNML*, which records the date and time of the compilation as well as the version of the compiler that produced this string.

The elements of the string have the following meanings:

YYYY
The year of the compilation

MM The month of the compilation

DD The day of the compilation

HH The hour of the compilation

MI The minute of the compilation

SS The second of the compilation

VN The version number of the compiler

RN The release number of the compiler

ML The maintenance level of the compiler

The timestamp can be located from the PPA2: at offset 12 in the PPA2 is a four-byte integer giving the offset (possibly negative) to the timestamp from the address of the PPA2.

The PPA2, in turn, can be located from the PPA1: at offset 4 in the PPA1 is a four-byte integer giving the offset (possibly negative) to the PPA2 from the entry point address corresponding to that PPA1.

You can locate the PPA1 as follows:

- If the code was compiled with LP(32), you can locate the PPA1 from the entry point address for a block: at offset 12 from the entry point address is a four-byte integer giving the offset (possibly negative) to the PPA1 from the entry point address.
- If the code was compiled with LP(64), the offset to the PPA1 is in the fullword eight bytes in front of each entry point address.

Saved options string

The compiler stores in the load module a 32-byte string that records the compiler options used in building the load module.

The declaration for the saved options string is provided in the include file `ibmvsos` in the samples data set SIBMZSAM.

For most of the fields in the structure, the meaning of the field is obvious given its name, but a few of the fields need some explanation:

- sos_words holds the number of the bytes in the structure divided by 4.
- sos_version is the version number for this structure. It is not a compiler version number.
- The size of the structure and what fields have been set depends on the version number.

The saved options string is located after the timestamp in one of two ways:

1. If the service option has been specified, the string specified in the service option follows immediately after the timestamp as a character varying string. Then the saved options string follows after the service string as a second character varying string.
2. If the service option has not been specified, the saved options string follows immediately after the timestamp as a character varying string.

The length of the varying string that holds the saved options string might be longer than the size of the saved options string itself.

The presence (or absence) of the service string is indicated in the PPA2 by the flag byte at decimal offset 20 in the PPA2: if the result of anding this byte with '20'bx is not zero, then the service string is present.

In some earlier releases of the PL/I compiler, the compiler did not place a saved options string in the load module. The presence (or absence) of the saved options string is indicated in the PPA2 by the flag byte at decimal offset 20 in the PPA2: if the result of anding this byte with '02'bx is not zero, the saved options string is present.

Chapter 22. Interrupts and attention processing

To enable a PL/I program to recognize attention interrupts, two operations must be possible:

- You must be able to create an interrupt. This is done in different ways depending upon both the terminal you use and the operating system.
- Your program must be prepared to respond to the interrupt. You can write an ON ATTENTION statement in your program so that the program receives control when the ATTENTION condition is raised.

  Note: If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following options:
  - The INTERRUPT option (supported only in TSO)
  - A TEST option other than NOTEST or TEST(NONE,NOSYM)

    Compiling this way causes INTERRUPT(ON) to be in effect, unless you explicitly specify INTERRUPT(OFF) in PLIXOPT.

You can find the procedure used to create an interrupt in the IBM instruction manual for the operating system and terminal that you are using.

There is a difference between the interrupt (the operating system recognized your request) and the raising of the ATTENTION condition.

An *interrupt* is your request that the operating system notify the running program. If a PL/I program was compiled with the INTERRUPT compile-time option, instructions are included that test an internal interrupt switch at discrete points in the program. The internal interrupt switch can be set if any program in the load module was compiled with the INTERRUPT compile-time option.

The internal switch is set when the operating system recognizes that an interrupt request was made. The execution of the special testing instructions (polling) raises the ATTENTION condition. If a debugging tool hook (or a CALL PLITEST) is encountered before the polling occurs, the debugging tool can be given control before the ATTENTION condition processing starts.

Polling ensures that the ATTENTION condition is raised between PL/I statements, rather than within the statements.

Figure 109 on page 478 shows a skeleton program, an ATTENTION ON-unit, and several situations where polling instructions will be generated. In the program, polling will occur at the following:

- LABEL1
- Each iteration of the DO
- The ELSE PUT SKIP ... statement
- Block END statements

© Copyright IBM Corp. 1999, 2017

```
%PROCESS INTERRUPT;
     .
     .
     .
 ON ATTENTION
   BEGIN;
     DCL X FIXED BINARY(15);
     PUT SKIP LIST ('Enter 1 to terminate, 0 to continue.');
     GET SKIP LIST (X);
     IF X = 1 THEN
       STOP;
     ELSE
       PUT SKIP LIST ('Attention was ignored');
   END;
     .
     .
     .
  LABEL1:
   IF EMPNO ...
     .
     .
     .
   DO I = 1 TO 10;
     .
     .
     .
   END;
     .
     .
     .
```

Figure 109. Using an ATTENTION ON-unit

Using ATTENTION ON-units

You can use processing within the ATTENTION ON-unit to terminate potentially endless looping in a program.

Control is given to an ATTENTION ON-unit when polling instructions recognize that an interrupt has occurred. Normal return from the ON-unit is to the statement following the polling code.

Interaction with a debugging tool

If the program has the TEST(ALL) or TEST(ERROR) runtime option in effect, an interrupt causes the debugging tool to receive control the next time a hook is encountered. This might be before the program's polling code recognizes that the interrupt occurred.

Later, when the ATTENTION condition is raised, the debugging tool receives control again for condition processing.

Chapter 23. Using the Checkpoint/Restart facility

This chapter describes the PL/I Checkpoint/Restart feature, which provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment.

At points specified in the program, information about the current status of the program is written as a record on a data set. If the program terminates due to a system failure, you can use this information to restart the program close to the point where the failure occurred, avoiding the need to rerun the program completely.

This restart can be either automatic or deferred. An automatic restart is one that takes place immediately (provided the operator authorizes it when requested by a system message). A deferred restart is one that is performed later as a new job.

You can request an automatic restart from within your program without a system failure having occurred.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system. This facility is described in the books listed in "Bibliography" on page 527.

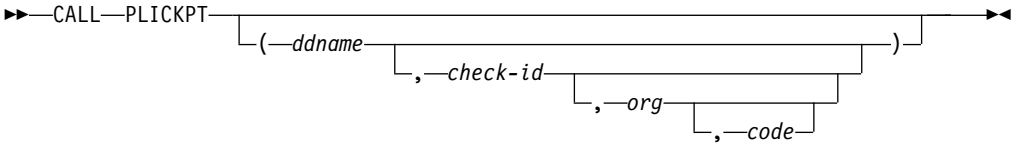
To use checkpoint/restart, you must do the following operations:

- Request, at suitable points in your program, that a checkpoint record is written. This is done with the built-in subroutine PLICKPT.
- Provide a data set on which the checkpoint record can be written.
- To ensure the desired restart activity, you might need to specify the RD parameter in the EXEC or JOB statement (see the *z/OS JCL Reference*).

Note: You should be aware of the restrictions affecting data sets used by your program. These are detailed in the "Bibliography" on page 527.

Requesting a checkpoint record

Each time you want a checkpoint record to be written, you must invoke, from your PL/I program, the built-in subroutine PLICKPT.



The four arguments are all optional. If you do not use an argument, you need not specify it unless you specify another argument that follows it in the given order. In this case, you must specify the unused argument as a null string ('').

ddname
Is a character string constant or variable specifying the name of the DD statement defining the data set that is to be used for checkpoint records. If you omit this argument, the system will use the default ddname SYSCHK.

© Copyright IBM Corp. 1999, 2017

check-id
Is a character string constant or variable specifying the name that you want to assign to the checkpoint record so that you can identify it later. If you omit this argument, the system will supply a unique identification and print it at the operator's console.

org
Is a character string constant or variable with the attributes CHARACTER(2) whose value indicates, in operating system terms, the organization of the checkpoint data set. You can specify the following values:

PS Indicates sequential (that is, CONSECUTIVE) organization.

PO Represents partitioned organization.

If you omit this argument, PS is assumed.

code
Is a variable with the attributes FIXED BINARY (31), which can receive a return code from PLICKPT. The return code has the following values:

0 A checkpoint has been successfully taken.

4 A restart has been successfully made.

8 A checkpoint has not been taken. The PLICKPT statement should be checked.

12 A checkpoint has not been taken. Check for a missing DD statement, a hardware error, or insufficient space in the data set. A checkpoint will fail if taken while a DISPLAY statement with the REPLY option is still incomplete.

16 A checkpoint has been taken, but ENQ macro calls are outstanding and will not be restored on restart. This situation will not normally arise for a PL/I program.

Defining the checkpoint data set

You must include a DD statement in the job control procedure to define the data set in which the checkpoint records are to be placed.

This data set can have either CONSECUTIVE or partitioned organization. You can use any valid ddname. If you use the ddname SYSCHK, you do not need to specify the ddname when invoking PLICKPT.

You must specify a data set name only if you want to keep the data set for a deferred restart. The I/O device can be any direct access device.

To obtain only the last checkpoint record, specify status as NEW (or OLD if the data set already exists). This will cause each checkpoint record to overwrite the previous one.

To retain more than one checkpoint record, specify status as MOD. This will cause each checkpoint record to be added after the previous one.

If the checkpoint data set is a library, “check-id” is used as the member-name. Thus a checkpoint will delete any previously taken checkpoint with the same name.

For direct access storage, you should allocate enough primary space to store as many checkpoint records as you will retain. You can specify an incremental space allocation, but it will not be used. A checkpoint record is approximately 5000 bytes longer than the area of main storage allocated to the step.

No DCB information is required, but you can include any of the following, where applicable:

```
OPTCD=W, OPTCD=C, RECFM=UT
```

For information about these subparameters, see the *z/OS MVS JCL User's Guide*.

Requesting a restart

A restart can be automatic or deferred.

You can make automatic restarts after a system failure or from within the program itself. The system operator must authorize all automatic restarts when requested by the system.

Automatic restart after a system failure

If a system failure occurs after a checkpoint has been taken, the automatic restart will occur at the last checkpoint if you have specified RD=R (or omitted the RD parameter) in the EXEC or JOB statement.

If a system failure occurs before any checkpoint has been taken, an automatic restart, from the beginning of the job step, can still occur if you have specified RD=R in the EXEC or JOB statement.

After a system failure occurs, you can still force automatic restart from the beginning of the job step by specifying RD=RNC in the EXEC or JOB statement. By specifying RD=RNC, you are requesting an automatic step restart without checkpoint processing if another system failure occurs.

Automatic restart within a program

You can request a restart at any point in your program.

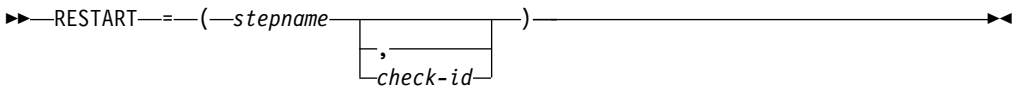
The rules for the restart are the same as for a restart after a system failure. To request the restart, you must execute the statement:

```
CALL PLIREST;
```

To effect the restart, the compiler terminates the program abnormally, with a system completion code of 4092. Therefore, to use this facility, the system completion code 4092 must not have been deleted from the table of eligible codes at system generation.

Getting a deferred restart

To ensure that automatic restart activity is canceled, but that the checkpoints are still available for a deferred restart, specify RD=NR in the EXEC or JOB statement when the program is first executed.



If you subsequently require a deferred restart, you must submit the program as a new job, with the RESTART parameter in the JOB statement. Use the RESTART parameter to specify the job step at which the restart is to be made and, if you want to restart at a checkpoint, the name of the checkpoint record.

For a restart from a checkpoint, you must also provide a DD statement that defines the data set containing the checkpoint record. The DD statement must be named SYSCHK. The DD statement must occur immediately before the EXEC statement for the job step.

Modifying checkpoint/restart activity

You can cancel automatic restart activity from any checkpoints taken in your program.

To cancel automatic restart, execute this statement:

```
CALL PLICANC;
```

However, if you specified RD=R or RD=RNC in the JOB or EXEC statement, automatic restart can still take place from the beginning of the job step.

Also, any checkpoints already taken are still available for a deferred restart.

You can cancel any automatic restart and the taking of checkpoints, even if they were requested in your program, by specifying RD=NC in the JOB or EXEC statement.

Chapter 24. Using user exits

PL/I provides a number of user exits that you can use to customize the PL/I product to suit your needs.

The PL/I products supply default exits and the associated source files.

If you want the exits to perform functions that are different from those supplied by the default exits, it is recommended that you modify the supplied source files as appropriate.

At times, it is useful to be able to tailor the compiler to meet the needs of your organization. For example, you might want to suppress certain messages or alter the severity of others. You might want to perform a specific function with each compilation, such as logging statistical information about the compilation into a file. A compiler user exit handles this type of function.

With PL/I, you can write your own user exit or use the exit provided with the product, either 'as is' or modified, depending on what you want to do with it. The user exit source code provided with the product can be seen in Figure 18 on page 176.

This chapter provides the following information:

- Procedures that the compiler user exit supports
- How to activate the compiler user exit
- IBMUEXIT, the IBM-supplied compiler user exit
- Requirements for writing your own compiler user exit

Procedures performed by the compiler user exit

The compiler user exit performs three specific procedures: initialization, interception and filtering of compiler messages, and termination.

As illustrated in Figure 110 on page 484, the compiler passes control to the initialization procedure, the message filter procedure, and the termination procedure. Each of these three procedures, in turn, passes control back to the compiler when the requested procedure is completed.

© Copyright IBM Corp. 1999, 2017

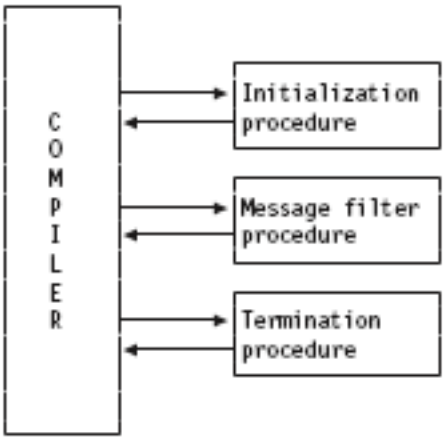


Figure 110. PL/I compiler user exit procedures

Each of the three procedures is passed two different control blocks:

- A *global control block* that contains information about the compilation. This is passed as the first parameter.
- A *function-specific control block* that is passed as the second parameter. The content of this control block depends upon which procedure has been invoked. For detailed information, see "Writing the initialization procedure" on page 487, "Writing the message filtering procedure" on page 487, and "Writing the termination procedure" on page 489.

Related information:

"Structure of global control blocks"
The global control block is passed to each of the three user exit procedures (initialization, filtering, and termination) whenever they are invoked.

Structure of global control blocks

The global control block is passed to each of the three user exit procedures (initialization, filtering, and termination) whenever they are invoked.

The following code and accompanying explanations describe the contents of each field in the global control block.

```
Dcl
  1 Uex_UIB           native based( null() ),
    2 Uex_UIB_Length         fixed bin(31),

    2 Uex_UIB_Exit_token     pointer,        /* for user exit's use */

    2 Uex_UIB_User_char_str  pointer,        /* to exit option str  */
    2 Uex_UIB_User_char_len  fixed bin(31),

    2 Uex_UIB_Filename_str   pointer,        /* to source filename  */
    2 Uex_UIB_Filename_len   fixed bin(31),

    2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs   */
    2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs   */

    2 Uex_UIB_Exit_Routs,                    /* exit entries set at
                                                initialization     */
```

```
3 ( Uex_UIB_Termination,
    Uex_UIB_Message_Filter,            /* call for each msg   */
    *, *, *, * )
  limited entry (
      *,                                /* to Uex_UIB          */
      *,                                /* to a request area   */
   );
```

Data Entry Fields

Uex_UIB_ Length
Contains the length of the control block in bytes. The value is `storage (Uex_UIB)`.

Uex_UIB_Exit_token
Used by the user exit procedure. For example, the initialization might set it to a data structure that is used by both the message filter and the termination procedures.

Uex_UIB_User_char_str
Points to an optional character string, if you specify it. For example, `pli filename (EXIT ('string'))...fn` can be a character string up to thirty-one characters in length.

Uex_UIB_char_len
Contains the length of the string pointed to by the `User_char_str`. The compiler sets this value.

Uex_UIB_Filename_str
Contains the name of the source file that you are compiling, and includes the drive and subdirectories as well as the filename. The compiler sets this value.

Uex_UIB_Filename_len
Contains the length of the name of the source file pointed to by the `Filename_str`. The compiler sets this value.

Uex_UIB_return_code
Contains the return code from the user exit procedure. The user sets this value.

Uex__UIB_reason_code
Contains the procedure reason code. The user sets this value.

Uex_UIB_Exit_Routs
Contains the exit entries set up by the initialization procedure.

Uex_UIB_Termination
Contains the entry that is to be called by the compiler at termination time. The user sets this value.

Uex_UIB_Message_Filter
Contains the entry that is to be called by the compiler whenever a message needs to be generated. The user sets this value.

The IBM-supplied compiler exit, IBMUEXIT

IBM supplies you with the sample compiler user exit, IBMUEXIT, which filters messages for you.

This compiler exit monitors messages and, based on the message number that you specify, suppresses the message or changes the severity of the message.

See the source of IBMUEXIT in Figure 18 on page 176.

Activating the compiler user exit

To activate the compiler user exit, you must specify the EXIT compile-time option.

The EXIT compile-time option allows you to specify a user-option-string that specifies the DDname for the user exit input file. If you do not specify a string, SYSUEXIT is used as the DDname for the user exit input file.

The user-option-string is passed to the user exit functions in the global control block. For additional information, see "Uex_UIB_User_char_str" in "Structure of global control blocks" on page 484.

Related information:

"EXIT" on page 33
The EXIT option enables the compiler user exit to be invoked.

Customizing the compiler user exit

You can write your own compiler user exit or simply use the one supplied with the compiler. In either case, the name of the fetchable file for the compiler user exit must be IBMUEXIT.

This section describes how to complete the following tasks:
- Modify the user exit input file for customized message filtering.
- Create your own compiler user exit.

Modifying SYSUEXIT

Rather than spending the time to write a completely new compiler user exit, you can simplify modify the user exit input file.

Edit the file to indicate which message numbers you want to suppress, and which message number severity levels you want to change. A sample file is shown in Figure 111.

```
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+--------------------------------
  'IBM'     1042       -1          1      String spans multiple lines
  'IBM'     1044       -1          1      FIXED BIN 7 mapped to 1 byte
  'IBM'     1047        8          0      Order inhibits optimization
  'IBM'     1052       -1          1      Nodescriptor with * extent arg
  'IBM'     1059        0          0      Select without OTHERWISE
  'IBM'     1169        0          1      Precision of result determined
```

Figure 111. Example of an user exit input file

The first two lines are header lines and are ignored by IBMUEXIT. The remaining lines contain input separated by a variable number of blanks.

Each column of the file is relevant to the compiler user exit:
- The first column should contain the letters `'IBM'` in single quotation marks for all compiler messages to which you want the exit to apply.
- The second column contains the four digit message number.
- The third column shows the new message severity. Severity -1 indicates that the severity should be left as the default value.

- The fourth column indicates whether the message is to be suppressed. Specify one of the following values:

  1 Suppresses the message.

  0 Prints the message.
- The comment field, found in the last column, is for your information, and is ignored by IBMUEXIT.

Writing your own compiler exit

To write your own user exit, you can use IBMUEXIT as a model. When you write the exit, make sure it covers the areas of initialization, message filtering, and termination.

See the source of IBMUEXIT in Figure 18 on page 176.

The compiler user exit must be compiled with the RENT option and linked as a DLL.

Writing the initialization procedure

Your initialization procedure should perform any initialization required by the exit, such as opening files and allocating storage.

Code the initialization procedure-specific control block as follows:

```
Dcl 1 Uex_ISA native based( null() ),
    2 Uex_ISA_Length_fixed bin(31);   /* storage(Uex_ISA) *  /
```

For information about the global control block syntax for the initialization procedure, see “Structure of global control blocks” on page 484.

Upon completion of the initialization procedure, you should set the return/reason codes to the following:

0/0
Continue compilation

4/n
Reserved for future use

8/n
Reserved for future use

12/n
Reserved for future use

16/n
Abort compilation

Writing the message filtering procedure

The message filtering procedure permits you to either suppress messages or alter the severity of messages.

You can increase the severity of any of the messages but you can decrease the severity only of **ERROR** (severity code 8) or **WARNING** (severity code 4) messages.

The procedure-specific control block contains information about the messages. It is used to pass information back to the compiler indicating how a particular message should be handled.

The following example shows a procedure-specific message filter control block:

```
Dcl 1 Uex_MFX native based( null() ),
      2 Uex_MFX_Length   fixed bin(31),

      2 Uex_MFX_Facility_Id  char(3),        /* of component writing
                                               message             */
      2 *                    char(1),
      2 Uex_MFX_Message_no   fixed bin(31),
      2 Uex_MFX_Severity     fixed bin(15),
      2 Uex_MFX_New_Severity fixed bin(15),  /* set by exit proc    */
      2 Uex_MFX_Inserts                fixed bin(15),
      2 Uex_MFX_Inserts_Data( 6 refer(Uex_MFX_Inserts) ),
        3 Uex_MFX_Ins_Type             fixed bin(7),
        3 Uex_MFX_Ins_Type_Data union unaligned,
          4 *                          char(8),
          4 Uex_MFX_Ins_Bin8           fixed bin(63),
          4 Uex_MFX_Ins_Bin            fixed bin(31),
          4 Uex_MFX_Ins_Str,
            5 Uex_MFX_Ins_Str_Len      fixed bin(15),
            5 Uex_MFX_Ins_Str_Addr     pointer,
          4 Uex_MFX_Ins_Series,
            5 Uex_MFX_Ins_Series_Sep   char(1),
            5 Uex_MFX_Ins_Series_Addr  pointer;
```

Data Entry Fields

Uex_MFX_Length
Contains the length of the control block in bytes. The value is storage (Uex_MFX).

Uex_MFX_Facility_Id
Contains the ID of the facility; for the compiler, the ID is `IBM`. The compiler sets this value.

Uex_MFX_Message_no
Contains the message number that the compiler is going to generate. The compiler sets this value.

Uex_MFX_Severity
Contains the severity level of the message; it can be from one to fifteen characters in length. The compiler sets this value.

Uex_MFX_New_Severity
Contains the new severity level of the message; it can be from one to fifteen characters in length. The user sets this value.

Uex_MFX_Inserts
Contains the number of inserts for the message; it can range from zero to six. The compiler sets this value.

Uex_MFX_Inserts_Data
Contains fields to describe each of the inserts. The compiler sets these values.

Uex_MFX_Ins_Type
Contains the type of the insert. These are the possible insert types:

Uex_Ins_Type_Xb31
Is used for a FIXED BIN(31) and has the value 1.

Uex_Ins_Type_Char
Is used for a CHAR string and has the value 2.

Uex_Ins_Type_Series
Is used for a series of CHAR strings and has the value 3.

Uex_Ins_Type_Xb63
Is used for a FIXED BIN(63) and has the value 4.

The compiler sets this value.

Uex_MFX_Ins_Bin
Contains the integer value for an insert that has integer type. The compiler sets this value.

Uex_MFX_Ins_Str_Len
Contains the length (in bytes) for an insert that has character type. The compiler sets this value.

Uex_MFX_Ins_Str_Addr
Contains the address of the character string for an insert that has character type. The compiler sets this value.

Uex_MFX_Ins_Series_Sep
Contains the character that should be inserted between each element for an insert that has series type. Typically, this is a blank, period, or comma. The compiler sets this value.

Uex_MFX_Ins_Series_Addr
Contains the address of the series of varying character strings for an insert that has series type. The address points to a FIXED BIN(31) field holding the number of strings to concatenate followed by the addresses of those strings. The compiler sets this value.

Upon completion of the message filtering procedure, set the return/reason codes to one of the following:

0/0
Continue compilation, output message

0/1
Continue compilation, do not output message

4/n
Reserved for future use

8/n
Reserved for future use

16/n
Abort compilation

Writing the termination procedure

You should use the termination procedure to perform any cleanup required, such as closing files. You might also want to write out final statistical reports based on information collected during the error message filter procedures and the initialization procedures.

Code the termination procedure-specific control block as follows:

```
Dcl 1 Uex_ISA native based,
    2 Uex_ISA_Length_fixed bin(31); /* storage(Uex_ISA)       */
```

For information about the global control block syntax for the termination procedure, see "Structure of global control blocks" on page 484. Upon completion of the termination procedure, set the return/reason codes to one of the following:

0/0
Continue compilation

4/n
Reserved for future use

8/n
Reserved for future use

12/n
Reserved for future use

16/n
Abort compilation

Example of suppressing SQL messages

This example shows how to modify the user exit to examine the message inserts and suppress two SQL informational messages and one SQL warning message.

```
*Process dft(nodescriptor connected);
*Process or('|') not('!');
*Process limits(extname(31)) rent;

 /*********************************************************************/
 /*                                                                   */
 /*  NAME - IBMUEXIT.PLI                                              */
 /*                                                                   */
 /*  DESCRIPTION                                                      */
 /*    User-exit sample program.                                      */
 /*                                                                   */
 /*    Licensed Materials - Property of IBM                           */
 /*    5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,2011.          */
 /*    All Rights Reserved.                                           */
 /*    US Government Users Restricted Rights-- Use, duplication or    */
 /*    disclosure restricted by GSA ADP Schedule Contract with        */
 /*    IBM Corp.                                                      */
 /*                                                                   */
 /*  DISCLAIMER OF WARRANTIES                                         */
 /*    The following enclosed code is sample code created by IBM      */
 /*    Corporation. This sample code is not part of any standard      */
 /*    IBM product and is provided to you solely for the purpose of   */
 /*    assisting you in the development of your applications.  The    */
 /*    code is provided "AS IS", without warranty of any kind.        */
 /*    IBM shall not be liable for any damages arising out of your    */
 /*    use of the sample code, even if IBM has been advised of the    */
 /*    possibility of such damages.                                   */
 /*                                                                   */
 /*********************************************************************/
```

Figure 112. Suppressing SQL messages

```
/*********************************************************************/
/*                                                                   */
/*  During initialization, IBMUEXIT is called.  It reads             */
/*  information about the messages being screened from a text        */
/*  file and stores the information in a linked list. IBMUEXIT       */
/*  also sets up the entry points for the message filter service     */
/*  and termination service.                                         */
/*                                                                   */
/*  For each message generated by the compiler, the compiler         */
/*  calls the message filter registered by IBMUEXIT.  The filter     */
/*  looks the message up in the linked list previously created       */
/*  to see if it is one for which some action should be taken.       */
/*                                                                   */
/*  The termination service is called at the end of the compile      */
/*  but does nothing.  It could be enhanced to generates reports     */
/*  or do other cleanup work.                                        */
/*                                                                   */
/*********************************************************************/


pack: package exports(*);

  Dcl
    1 Uex_UIB           native Based( null() ),
      2 Uex_UIB_Length         fixed bin(31),

      2 Uex_UIB_Exit_token     pointer,       /* for user exit's use */

      2 Uex_UIB_User_char_str  pointer,       /* to exit option str  */
      2 Uex_UIB_User_char_len  fixed bin(31),

      2 Uex_UIB_Filename_str   pointer,       /* to source filename  */
      2 Uex_UIB_Filename_len   fixed bin(31),

      2 Uex_UIB_return_code fixed bin(31),    /* set by exit procs   */
      2 Uex_UIB_reason_code fixed bin(31),    /* set by exit procs   */

      2 Uex_UIB_Exit_Routs,                   /* exit entries setat
                                                 initialization      */
        3 ( Uex_UIB_Termination,
            Uex_UIB_Message_Filter,           /* call for each msg   */
           *, *, *, * )
          limited entry (
           *,                                 /* to Uex_UIB          */
           *                                  /* to a request area   */
          );

  /*******************************************************************/
  /*                                                                 */
  /*   request area for initialization exit                          */
  /*                                                                 */
  /*******************************************************************/

  Dcl 1 Uex_ISA native based( null() ),
       2 Uex_ISA_Length fixed bin(31);
```

Suppressing SQL messages (continued)

```
  /*****************************************************************/
  /*                                                               */
  /*   request area for message_filter exit                        */
  /*                                                               */
  /*****************************************************************/

  Dcl 1 Uex_MFX based( null() ),
       2 Uex_MFX_Length                 fixed bin(31),
       2 Uex_MFX_Facility_Id            char(3),
       2 Uex_MFX_Version                fixed bin(7),
       2 Uex_MFX_Message_no             fixed bin(31),
       2 Uex_MFX_Severity               fixed bin(15),
       2 Uex_MFX_New_Severity           fixed bin(15),
       2 Uex_MFX_Inserts                fixed bin(15),
       2 Uex_MFX_Inserts_Data( 6 ),
         3 Uex_MFX_Ins_Type             fixed bin(7),
         3 Uex_MFX_Ins_Type_Data union unaligned,
           4 *                          char(8),
           4 Uex_MFX_Ins_Bin            fixed bin(31),
           4 Uex_MFX_Ins_Str,
             5 Uex_MFX_Ins_Str_Len      fixed bin(15),
             5 Uex_MFX_Ins_Str_Addr     pointer,
          4 Uex_MFX_Ins_Series,
             5 Uex_MFX_Ins_Series_Sep   char(1),
             5 Uex_MFX_Ins_Series_Addr  pointer;

  dcl uex_Ins_Type_Xb31               fixed bin(15) value(1);
  dcl uex_Ins_Type_Char               fixed bin(15) value(2);
  dcl uex_Ins_Type_Series             fixed bin(15) value(3);

 /*****************************************************************/
  /*                                                               */
  /*   request area for terminate exit                             */
  /*                                                               */
  /*****************************************************************/

  Dcl 1 Uex_TSA native based( null() ),
        2 Uex_TSA_Length fixed bin(31);

  /*****************************************************************/
  /*                                                               */
  /*   severity codes                                              */
  /*                                                               */
  /*****************************************************************/

  dcl uex_Severity_Normal             fixed bin(15) value(0);
  dcl uex_Severity_Warning            fixed bin(15) value(4);
  dcl uex_Severity_Error              fixed bin(15) value(8);
  dcl uex_Severity_Severe             fixed bin(15) value(12);
  dcl uex_Severity_Unrecoverable      fixed bin(15) value(16);

  /*****************************************************************/
  /*                                                               */
  /*   return codes                                                */
  /*                                                               */
  /*****************************************************************/

  dcl uex_Return_Normal               fixed bin(15) value(0);
  dcl uex_Return_Warning              fixed bin(15) value(4);
  dcl uex_Return_Error                fixed bin(15) value(8);
  dcl uex_Return_Severe               fixed bin(15) value(12);
  dcl uex_Return_Unrecoverable        fixed bin(15) value(16);
```

Suppressing SQL messages (continued)

```
  /*****************************************************************/
  /*                                                               */
  /*   reason codes                                                */
  /*                                                               */
  /*****************************************************************/

  dcl uex_Reason_Output               fixed bin(15) value(0);
  dcl uex_Reason_Suppress             fixed bin(15) value(1);

  dcl header  pointer;

  dcl
    1 message_item native based,
      2 message_Info,
        3 facid     char(3),
        3 msgno     fixed bin(31),
        3 newsev    fixed bin(15),
        3 reason    fixed bin(31),
        3 variable  char(31) var,
      2 link pointer;

ibmuexit: proc ( ue, ia ) options( fetchable );

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 ia like uex_Isa byaddr;

  dcl sysuexit     file stream input env(recsize(80));
  dcl next         pointer;
  dcl based_Chars  char(8) based;
  dcl title_Str    char(8) var;
  dcl eof          bit(1);


  on error
    begin;
      on error system;
      call plidump('TFBHS' );
    end;

  on undefinedfile(sysuexit)
    begin;
      put edit ('** User exit unable to open exit file ')
               (A) skip;
      put skip;
      signal error;
    end;

  if ue.uex_Uib_User_Char_Len = 0 then
    do;
      open file(sysuexit);
    end;
  else
    do;
      title_Str
       = substr( ue.uex_Uib_User_Char_Str->based_Chars,
                 1, ue.uex_Uib_User_Char_Len );
      open file(sysuexit) title(title_Str);
    end;
```

Suppressing SQL messages (continued)

```
/*************************************************************/
/*                                                           */
/*  save the address of the message filter so that it will   */
/*  be invoked by the compiler                               */
/*                                                           */
/*************************************************************/

ue.Uex_UIB_Message_Filter = message_filter;

/*************************************************************/
/*                                                           */
/*  set the pointer to the linked list to null               */
/*                                                           */
/*  then allocate the first message record                   */
/*                                                           */
/*************************************************************/

header = sysnull();
allocate message_item set(next);

/*********************************************************************/
/*                                                                   */
/*  skip header lines and read the file                              */
/*                                                                   */
/*  the file is expected to start with a header line and             */
/*  then a line with a scale and then the data lines, for example,*/
/*  it could look like the 5 lines below starting with "Fac Id"   */
/*                                                                   */
/*   Fac Id   Msg No   Severity   Suppress   Insert                  */
/* +-------+-------+---------+---------+-----------------------  */
/*   'IBM'     3259        0          1      'DSNH527'             */
/*   'IBM'     3024        0          1      'DSNH4760'            */
/*   'IBM'     3024        0          1      'DSNH050'             */
/*                                                                   */
/*********************************************************************/

eof = '0'b;
on endfile(sysuexit)
  eof = '1'b;

get file(sysuexit) list(next->message_info) skip(3);

do while( eof = '0'b );

  /*************************************************************/
  /*                                                           */
  /*  put message information in linked list                   */
  /*                                                           */
  /*************************************************************/

  next->link = header;
  header = next;

  /*************************************************************/
  /*                                                           */
  /*  read next data line                                      */
  /*                                                           */
  /*************************************************************/

  allocate message_item set(next);
  get file(sysuexit) skip;
  get file(sysuexit) list(next->message_info);

end;
```

Suppressing SQL messages (continued)

```
  /*************************************************************/
  /*                                                           */
  /*  free the last message record allocated and close the file  */
  /*                                                           */
  /*************************************************************/

  free next->message_Item;
  close file(sysuexit);

end;

message_Filter: proc ( ue, mf  );

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 mf like uex_Mfx byaddr;

  dcl next         pointer;
  dcl jx           fixed bin(31);
  dcl insert       char(256) var;
  dcl based_Chars  char(256) based;

  on error
    begin;
      on error system;
      call plidump('TFBHS' );
    end;

  /*************************************************************/
  /*                                                           */
  /*  by default, leave the reason code etc unchanged          */
  /*                                                           */
  /*************************************************************/

  ue.uex_Uib_Reason_Code = uex_Reason_Output;
  ue.uex_Uib_Return_Code = 0;

  mf.uex_Mfx_New_Severity = mf.uex_Mfx_Severity;

  /*************************************************************/
  /*                                                           */
  /*  save the first insert if it has character type           */
  /*                                                           */
  /*************************************************************/

  insert = '*';
  if mf.Uex_MFX_Length < stg(mf) then;
  else
    if mf.Uex_MFX_Inserts = 0 then;
    else
      do jx = 1 to mf.Uex_MFX_Inserts;
        select( mf.Uex_MFX_Ins_Type(jx) );
          when( uex_Ins_Type_Char )
            do;
              if jx = 1 then
                insert =
                  substr( mf.Uex_MFX_Ins_Str_Addr(jx)->based_Chars,
                          1,mf.Uex_MFX_Ins_Str_Len(jx));
            end;
          otherwise;
        end;
      end;
```

Suppressing SQL messages (continued)

```
  /*************************************************************/
  /*                                                           */
  /*  search list for matching error message                   */
  /*                                                           */
  /*************************************************************/

  search_list:
  do next = header repeat( next->link ) while( next !=sysnull() );

    if next->msgno = mf.uex_Mfx_Message_No
     & next->facid = mf.Uex_Mfx_Facility_Id then
      do;
        if next->variable = '*' then
          leave search_list;
        if next->variable
         = substr(insert,1,length(next->variable)) then
          leave search_list;
      end;
  end;

  /*************************************************************/
  /*                                                           */
  /*  if list exhausted, then                                  */
  /*    no match was found                                     */
  /*  else                                                     */
  /*    filter the message according to the match found        */
  /*                                                           */
  /*************************************************************/

  if next = sysnull() then;
  else
    do;
      /*********************************************************/
      /*                                                       */
      /*  filter error based on information in table           */
      /*                                                       */
      /*********************************************************/

      ue.uex_Uib_Reason_Code = next->reason;
      if next->newsev < 0 then;
      else
        mf.uex_Mfx_New_Severity = next->newsev;
    end;
end;

exitterm: proc ( ue, ta  );

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 ta like uex_Tsa byaddr;

  ue.uex_Uib_return_Code = 0;
  ue.uex_Uib_reason_Code = 0;

end;

end;
```

Suppressing SQL messages (continued)

Chapter 25. PL/I descriptors

This chapter describes PL/I parameter passing conventions between PL/I routines at run time.

For additional information about Language Environment runtime environment considerations, other than descriptors, see the *z/OS Language Environment Programming Guide*. This includes runtime environment conventions and assembler macros supporting these conventions.

Passing an argument

When a string, an array, or a structure is passed as an argument, the compiler passes a descriptor for that argument unless the called routine is declared with OPTIONS(NODESCRIPTOR).

There are two methods for passing such descriptors:
- By descriptor list
- By descriptor locator

Note the following key features about each of these two methods:
- **When arguments are passed with a descriptor list**
  - The number of arguments passed is one greater than the number of arguments specified if any of the arguments needs a descriptor.
  - An argument passed with a descriptor can be received as a pointer passed by value (BYVALUE).
  - The compiler uses this method when the DEFAULT(DESCLIST) compiler option is in effect.
- **When arguments are passed by descriptor locator**
  - The number of arguments passed always matches the number of arguments specified.
  - An argument passed with a descriptor can be received as a pointer passed by reference (BYADDR).
  - The compiler uses this method when the DEFAULT(DESCLOCATOR) compiler option is in effect.

Argument passing by descriptor list

When arguments and their descriptors are passed with a descriptor list, an extra argument is passed whenever at least one argument needs a descriptor.

This extra argument is a pointer to a list of pointers. The number of entries in this list equals the number of arguments passed. For arguments that do not require a descriptor, the corresponding pointer in the descriptor list is set to SYSNULL. For arguments that do require a descriptor, the corresponding pointer in the descriptor list is set to the address of that argument's descriptor.

For example, suppose the routine `sample` is declared as follows:

```
declare sample entry( fixed bin(31), varying char(*) )
              options( byaddr descriptor );
```

© Copyright IBM Corp. 1999, 2017

Assume that sample is called as in the following statement:

```
call sample( 1, 'test' );
```

The following three arguments are passed to the routine:

- Address of a fixed bin(31) temporary with the value 1
- Address of a varying char(4) temporary with the value test
- Address of a descriptor list consisting of the following:
  - SYSNULL()
  - Address of the descriptor for a varying char(4) string

Argument passing by locator/descriptor

When arguments and their descriptors are passed by locator/descriptor, whenever an argument requires a descriptor, the address of a locator/descriptor for the argument is passed instead.

Except for strings, the locator/descriptor is a pair of pointers. The first pointer is the address of the data; the second pointer is the address of the descriptor. For strings, under CMPAT(LE), the locator/descriptor is still such a pair of pointers. But under the other CMPAT options, the locator/descriptor consists of the address of the string and then the string descriptor itself.

For example, suppose that the routine sample is declared as follows:

```
declare sample entry( fixed bin(31), varying char(*) )
              options( byaddr descriptor );
```

Assume that sample is called as in the following statement:

```
call sample( 1, 'test' );
```

The following two arguments are passed to the routine:

- The address of a fixed bin(31) temporary with the value 1
- The address of a locator/descriptor that consists of the following address and descriptor:
  - The address of a varying char(4) temporary with the value test
  - Under CMPAT(LE), the address of the CMPAT(LE) descriptor for a varying char(4) string
  - Under CMPAT(V*), the CMPAT(V*) descriptor for a varying char(4) string

CMPAT(V*) descriptors

Unlike LE descriptors, the CMPAT(V*) descriptors are not self-describing. However, the string descriptors are the same for all CMPAT(V*) options, and they also share the same codepage encoding as the LE string descriptors.

String descriptors

In a string descriptor, the first 2 bytes specify the maximum length for the string. This maximum length is always held in native format.

The third byte contains various flags (to indicate, for example, if the string length in a VARYING string is held in littleendian or bigendian format or if the data in a WIDECHAR string is held in littleendian or bigendian format).

In a string descriptor for a nonvarying bit string, the fourth byte gives the bit offset.

In a string descriptor for a CHARACTER string, the fourth byte encodes the compiler CODEPAGE option.

The declare for a string descriptor under CMPAT(V2) is as follows:

```
declare
  1 dso_string based( null() ),
    2 dso_string_length          fixed bin(15),
    2 dso_string_flags,
      3 dso_string_is_varying           bit(1),
      3 dso_string_is_varyingz          bit(1),
      3 dso_string_has_nonnative_len    bit(1), /* for varying     */
      3 dso_string_is_ascii             bit(1), /* for char        */
      3 dso_string_has_nonnative_data   bit(1), /* for wchar       */
      3 *                               bit(1), /* reserved, '0'b  */
      3 *                               bit(1), /* reserved, '0'b  */
      3 *                               bit(1), /* reserved, '0'b  */
    2 * union,
      3 dso_String_Codepage      ordinal ccs_Codepage_Enum,
      3 dso_string_bitofs        fixed bin(8) unsigned,
    2 dso_string_end     char(0);
```

The declare for a string descriptor under CMPAT(V3) is as follows:

```
declare
  1 dso_longstr based( null() ),
    2 dso_longstr_info,
      3 *                   fixed bin(8) unsigned,
      3 dso_datatype        fixed bin(8) unsigned,
      3 * union,
        4 dso_longstr_bitofs    fixed bin(8) unsigned,
        4 dso_longstr_codepage  type ccs_Codepage_Enum,
      3 dso_longstr_info2,
        4 dso_longstr_has_nonnative_len   bit(1), /* for varying    */
        4 dso_longstr_is_ebcdic           bit(1), /* for char       */
        4 dso_longstr_has_nonnative_data  bit(1), /* for wchar      */
        4 *                               bit(1),
        4 dso_longstr_is_varying          bit(1),
        4 dso_longstr_is_varyingz         bit(1),
        4 dso_longstr_is_varying4         bit(1),
        4 *                               bit(1),
    2 dso_longstr_length          fixed bin(31),
    2 dso_longstr_end     char(0);
```

The possible values for the codepage encoding are defined as follows:

```
define ordinal
  ccs_Codepage_Enum
    ( ccs_Codepage_01047  value(1)
     ,ccs_Codepage_01140
     ,ccs_Codepage_01141
     ,ccs_Codepage_01142
     ,ccs_Codepage_01143
     ,ccs_Codepage_01144
     ,ccs_Codepage_01145
     ,ccs_Codepage_01146
     ,ccs_Codepage_01147
     ,ccs_Codepage_01148
     ,ccs_Codepage_01149
     ,ccs_Codepage_00819
     ,ccs_Codepage_00813
     ,ccs_Codepage_00920
     ,ccs_Codepage_00037
     ,ccs_Codepage_00273
```

```
    ,ccs_Codepage_00277
    ,ccs_Codepage_00278
    ,ccs_Codepage_00280
    ,ccs_Codepage_00284
    ,ccs_Codepage_00285
    ,ccs_Codepage_00297
    ,ccs_Codepage_00500
    ,ccs_Codepage_00871
    ,ccs_Codepage_01026
    ,ccs_Codepage_01155
   ) unsigned prec(8);
```

Array descriptors

In the following declares, the upper bound for the arrays is declared as 15, but it should be understood that the actual upper bound will always match the number of dimensions in the array it describes.

The declare for a CMPAT(V2) array descriptor is as follows:

```
declare
  1 dso_v2  based( null() ),
    2 dso_v2_rvo      fixed bin(31),    /* relative virtual origin */
    2 dso_v2_data(1:15),
      3 dso_v2_stride fixed bin(31),    /*   multiplier            */
      3 dso_v2_hbound fixed bin(31),    /*   hbound                */
      3 dso_v2_lbound fixed bin(31);    /*   lbound                */
```

The declare for a CMPAT(V3) array descriptor is as follows:

```
declare
  1 dso_v3  based( null() ),
    2 dso_v3_rvo      fixed bin(63),    /* relative virtual origin */
    2 dso_v3_data(1:15),
      3 dso_v3_stride fixed bin(63),    /*   multiplier            */
      3 dso_v3_hbound fixed bin(63),    /*   hbound                */
      3 dso_v3_lbound fixed bin(63);    /*   lbound                */
```

CMPAT(LE) descriptors

Every LE descriptor starts with a 4-byte field. The first byte specifies the descriptor type (scalar, array, structure, or union). The remaining three bytes are zero unless they are set by the particular descriptor type.

The declare for a descriptor header is as follows:

```
declare
  1 dsc_Header based( sysnull() ),
    2 dsc_Type          fixed bin(8) unsigned,
    2 dsc_Datatype      fixed bin(8) unsigned,
    2 *                 fixed bin(8) unsigned,
    2 *                 fixed bin(8) unsigned;
```

These are possible values for the dsc_Type field:

```
declare
  dsc_Type_Unset               fixed bin(8) value(0),
  dsc_Type_Element             fixed bin(8) value(2),
  dsc_Type_Array               fixed bin(8) value(3),
  dsc_Type_Structure           fixed bin(8) value(4),
  dsc_Type_Union               fixed bin(8) value(4);
```

String descriptors

In a string descriptor, the second byte of the header indicates the string type (bit, character or graphic as well as nonvarying, varying or varyingz).

In a string descriptor for a nonvarying bit string, the third byte of the header gives the bit offset.

In a string descriptor for a CHARACTER string, the third byte of the header encodes the compiler CODEPAGE option.

In a string descriptor for a varying string, the fourth byte has a bit indicating if the string length is held in nonnative format.

In a string descriptor for a character string, the fourth byte also has a bit indicating if the string data is in EBCDIC.

The declare for a string descriptor is as follows:

```
declare
  1 dsc_String  based( sysnull() ),
    2 dsc_String_Header,
      3 *                     fixed bin(8) unsigned,
      3 dsc_String_Type       fixed bin(8) unsigned,
      3 * union,
        4 dsc_String_Codepage ordinal ccs_Codepage_Enum,
        4 dsc_String_BitOfs   fixed bin(8) unsigned,
      3 *,
        4 dsc_String_Has_Nonnative_Len    bit(1),
        4 dsc_String_Is_Ebcdic            bit(1),
        4 dsc_String_Has_Nonnative_Data   bit(1),
        4 *                               bit(5),
    2 dsc_String_Length   fixed bin(31); /* max length of string */
```

These are the possible values for the dsc_String_Type field:

```
declare
  dsc_String_Type_Unset                fixed bin(8) value(0),
  dsc_String_Type_Char_Nonvarying      fixed bin(8) value(2),
  dsc_String_Type_Char_Varyingz        fixed bin(8) value(3),
  dsc_String_Type_Char_Varying2        fixed bin(8) value(4),
  dsc_String_Type_Bit_Nonvarying       fixed bin(8) value(6),
  dsc_String_Type_Bit_Varying2         fixed bin(8) value(7),
  dsc_String_Type_Graphic_Nonvarying   fixed bin(8) value(9),
  dsc_String_Type_Graphic_Varyingz     fixed bin(8) value(10),
  dsc_String_Type_Graphic_Varying2     fixed bin(8) value(11),
  dsc_String_Type_Widechar_Nonvarying  fixed bin(8) value(13),
  dsc_String_Type_Widechar_Varyingz    fixed bin(8) value(14),
  dsc_String_Type_Widechar_Varying2    fixed bin(8) value(15);
```

Array descriptors

The declare for an array descriptor is as follows:

```
declare
  1 dsc_Array  based( sysnull() ),
    2 dsc_Array_Header   like dsc_Header,
    2 dsc_Array_EltLen   fixed bin(31), /* Length of array element */
    2 dsc_Array_Rank     fixed bin(31), /* Count of dimensions     */
    2 dsc_Array_RVO      fixed bin(31), /* Relative virtual origin */
    2 dsc_Array_Data( 1: 1 refer(dsc_Array_Rank) ),
      3 dsc_Array_LBound fixed bin(31), /*   LBound                */
      3 dsc_Array_Extent fixed bin(31), /*   HBound - LBound + 1   */
      3 dsc_Array_Stride fixed bin(31); /*   Multiplier            */
```

Part 6. Appendixes

© Copyright IBM Corp. 1999, 2017

Appendix. SYSADATA message information

When you specify the MSG suboption of the XINFO compile-time option, the compiler generates a SYSADATA file.

The SYSADATA file contains the following records:

- Counter records
- Literal records
- File records
- Message records
- Options record

The full set of declarations for all the ADATA records are available in the members `ibmwxin` and `ibmwxop` in the samples data set SIBMZSAM.

You should note that the records in the file are not necessarily produced in the order listed above; for example, literal and file records might be interleaved. If you are writing code that reads a SYSADATA file, you should not rely on the order of the records in the file except for the following exceptions:

- Counter records are the first records in the file.
- Each literal record precedes any reference to the literal it defines.
- Each file record precedes any reference to the file it describes.

Understanding the SYSADATA file

The SYSADATA file is a sequential binary file.

Under z/OS batch, the compiler writes the SYSADATA records to the file specified by the SYSADATA DD statement, and that file must not be a member of a PDS. On all other systems, the compiler writes to a file with the extension `adt`.

Each record in the file contains a header. This 12-byte header has fields that are the same for all records in the file:

`Compiler`
A number representing the compiler that produced the data. For PL/I, the number is 40.

`Edition number`
The edition number of the compiler that produced the data. For this product, it is the number 2.

`SYSADATA level`
A number representing the level of SYSADATA that this file format represents. For this product, it is the number 4.

The header also has some fields that vary from record to record:

- Record type
- Whether the record is continued onto the next record

© Copyright IBM Corp. 1999, 2017

Possible record types are encoded as an ordinal value as shown in Figure 113.

```
Define ordinal xin_Rect
 (Xin_Rect_Msg       value(50),   /*  Message record               */
  Xin_Rect_Fil       value(57),   /*  File record                  */
  Xin_Rect_Opt       value(60),   /*  Options record               */
  Xin_Rect_Sum       value(61),   /*  Summary record               */
  Xin_Rect_Rep       value(62),   /*  Replace record               */
  Xin_Rect_Src       value(63),   /*  Source record                */
  Xin_Rect_Tok       value(64),   /*  Token record                 */
  Xin_Rect_Sym       value(66),   /*  Symbol record                */
  Xin_Rect_Lit       value(67),   /*  Literal record               */
  Xin_Rect_Syn       value(69),   /*  Syntax record                */
  Xin_Rect_Ord_Type  value(80),   /*  Ordinal type record          */
  Xin_Rect_Ord_Elem  value(81),   /*  Ordinal element record       */
  Xin_Rect_Ctr       value(82) )  /*  Counter record               */
                     prec(15);
```

Figure 113. Record types encoded as an ordinal value

The declare for the header part of a record is shown in Figure 114.

```
Dcl
  1 Xin_Hdr Based( null() ),       /*  Header portion               */
                                   /*                               */
    2 Xin_Hdr_Prod                 /*  Language code                */
            fixed bin(8) unsigned, /*                               */
                                   /*                               */
    2 Xin_Hdr_Rect                 /*  Record type                  */
        unal ordinal xin_Rect,     /*                               */
                                   /*                               */
    2 Xin_Hdr_Level                /*  SYSADATA level               */
            fixed bin(8) unsigned, /*                               */
                                   /*                               */
    2 * union,                     /*                               */
      3 xin_Hdr_Flags     bit(8),  /*  flags                        */
      3 *,                         /*                               */
        4 *               bit(6),  /*  Reserved                     */
        4 Xin_Hdr_Little_Endian    /*  ints are little endian       */
                          bit(1),  /*                               */
        4 Xin_Hdr_Cont    bit(1),  /*  Record continued in next rec */
                                   /*                               */
    2 Xin_Hdr_Edition              /*  compiler "edition"           */
            fixed bin(8) unsigned, /*                               */
                                   /*                               */
    2 Xin_Hdr_Fill        bit(32), /*  reserved                     */
                                   /*                               */
    2 Xin_Hdr_Data_Len             /*  length of data part          */
           fixed bin(16) unsigned, /*                               */
                                   /*                               */
    2 Xin_Hdr_End         char(0); /*                               */
```

Figure 114. Declare for the header part of a record

Summary record

The record with type Xin_Rect_Sum is the summary record and is the first record in the file.

Its declare is shown in Figure 115.

```
Dcl
  1 Xin_Sum     Based( null() ),   /* summary record               */
                                   /*                              */
    2 Xin_Sum_Hdr                  /* standard header              */
           like Xin_Hdr,           /*                              */
                                   /*                              */
    2 Xin_Sum_Max_Severity         /* max severity from compiler   */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Sum_Left_Margin          /* left margin                  */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Sum_Right_Margin         /* right margin                 */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 xin_Sum_Rsrvd(15)            /* reserved                     */
           fixed bin(32) unsigned; /*                              */
```

Figure 115. Declare for a summary record

Options record

The record with type Xin_Rect_Opt is the options record. It lists all the compiler options that are in effect during compilation.

The declaration for the options record is provided in the member `ibmwxop` in the samples data set SIBMZSAM.

Counter records

Each counter record specifies, for a subsequent record type, how many records of that type the file contains and how many bytes those records occupy.

```
Dcl
  1 xin_Ctr     Based( null() ),   /* counter/size record          */
                                   /*                              */
    2 xin_Ctr_Hdr                  /* standard header              */
           like xin_Hdr,           /*                              */
                                   /*                              */
    2 xin_Ctr_Rect                 /* record type                  */
        unal ordinal xin_Rect,     /*                              */
                                   /*                              */
    2 *                            /*                              */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 xin_Ctr_Count                /* count of that record type    */
           fixed bin(31) unsigned, /*                              */
                                   /*                              */
    2 xin_Ctr_Size                 /* size used                    */
           fixed bin(31) unsigned; /*                              */
```

Figure 116. Declare for a counter record

Literal records

Each literal record assigns a number, called a literal index, that is used by later records to refer to the characters named in this particular record.

```
Dcl
 1 xin_Lit     Based( null() ),   /* literal record               */
                                  /*                              */
    2 xin_Lit_Hdr                 /* standard header              */
           like xin_Hdr,          /*                              */
                                  /*                              */
    2 xin_Lit_Inx                 /* adata index for literal      */
           fixed bin(31) unsigned, /*                             */
                                  /*                              */
    2 xin_Lit_Len                 /* length of literal            */
           fixed bin(31) unsigned, /*                             */
                                  /*                              */
    2 xin_Lit_Val      char(2000); /* literal value               */
```

Figure 117. Declare for a literal record

File records

Each file record assigns a number, called a file index, that is used by later records to refer to the file described by this record. The described file might be the primary PL/I source file or an INCLUDEd file. Each file record specifies a literal index for the fully qualified name of the file.

For an INCLUDEd file, each file record also contains the file index and source line number from where the INCLUDE request came. (For primary source files, these fields are zero.)

```
Dcl
  1 xin_Fil     Based( null() ),   /* file record                  */
                                   /*                              */
    2 xin_Fil_Hdr                  /* standard header              */
           like xin_Hdr,           /*                              */
                                   /*                              */
    2 xin_Fil_File_Id              /* file id from whence it       */
           fixed bin(31) unsigned, /*  was INCLUDEd                */
                                   /*                              */
    2 xin_Fil_Line_No              /* line no within that file     */
           fixed bin(31) unsigned, /*                              */
                                   /*                              */
    2 xin_Fil_Id                   /* id assigned to this file     */
           fixed bin(31) unsigned, /*                              */
                                   /*                              */
    2 xin_Fil_Name                 /* literal index of the         */
           fixed bin(31) unsigned; /*  fully qualified file name   */
```

Figure 118. Declare for a file record

Message records

Each message record describes a message issued during the compilation. Message records are not generated for suppressed messages.

Each message record contains the following data:

- The file index and source line number for the file and line to which the message is attributed. If the message pertains to the compilation as a whole, these fields are zero.
- The identifier (for example, IBM1502) and severity associated with the message.

- The text of the message.

The declare for a message record is as follows:

```
Dcl
  1 xin_Msg     Based( null() ),   /* message record               */
                                   /*                              */
    2 xin_Msg_Hdr                  /* standard header              */
          like xin_Hdr,            /*                              */
                                   /*                              */
    2 xin_Msg_File_Id              /* file id                      */
          fixed bin(31) unsigned, /*                              */
                                   /*                              */
    2 xin_Msg_Line_No              /* line no within file          */
          fixed bin(31) unsigned, /*                              */
                                   /*                              */
    2 xin_Msg_Id                   /* identifier (i.e. IBM1502)    */
                        char(16), /*                              */
                                   /*                              */
    2 xin_Msg_Severity             /* severity (0, 4, 8, 12 or 16) */
          fixed bin(15)   signed, /*                              */
                                   /*                              */
    2 xin_Msg_Length               /* length of message            */
          fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Msg_Text                 /* actual message               */
        char( 100 refer(xin_Msg_Length) );
```

Figure 119. Declare for a message record

Understanding SYSADATA symbol information

When you specify the SYM suboption of the XINFO compile-time option, the compiler generates a SYSADATA file that contains symbol information in addition to the records generated for the MSG suboption.

The following records contain symbol information:

- Ordinal type records
- Ordinal element records
- Symbol records

Symbol records are not generated for built-in functions, generic variables, or variables with non-constant extents.

Ordinal type records

Each ordinal type record assigns a number, called an ordinal type index, that is used by later records to refer to an ordinal type described by this record. The name of the type is indicated by a literal index. Each ordinal type record contains the file index and source line number for the file and line in which the ordinal type was declared.

Each ordinal type record contains:

- A count of the number of values defined by that type
- The precision associated with the type
- Bits indicating if it is signed or unsigned

```
declare                              /*                              */
  1 xin_Ord_Type   based( null() ), /*                              */
                                     /*                              */
    2 xin_Ord_Type_Hdr               /* standard header              */
          like xin_Hdr,              /*                              */
                                     /*                              */
    2 xin_Ord_Type_File_Id           /* file id                      */
          fixed bin(31) unsigned,    /*                              */
                                     /*                              */
    2 xin_Ord_Type_Line_No           /* line no within file          */
          fixed bin(31) unsigned,    /*                              */
                                     /*                              */
    2 xin_Ord_Type_Id                /* identifying number           */
                  fixed bin(31),     /*                              */
                                     /*                              */
    2 xin_Ord_Type_Count             /* count of elements            */
                  fixed bin(31),     /*                              */
                                     /*                              */
    2 xin_Ord_Type_Prec              /* precision for ordinal        */
          fixed bin(08) unsigned,    /*                              */
                                     /*                              */
    2 *,                             /*                              */
      3 xin_Ordinal_Type_Signed      /* signed attribute applies     */
                        bit(1),      /*                              */
      3 xin_Ordinal_Type_Unsigned    /* unsigned attribute applies   */
                        bit(1),      /*                              */
      3 *                            /* unused                       */
                        bit(6),      /*                              */
                                     /*                              */
    2 *                              /* unused                       */
                       char(2),      /*                              */
                                     /*                              */
    2 xin_Ord_Type_Name              /* type name                    */
                  fixed bin(31);     /*                              */
```

Figure 120. Declare for an ordinal type record

Ordinal element records

Each ordinal type record is immediately followed by a series of records (as many as specified by the ordinal type count) that describes the values named by that ordinal. Each ordinal element record assigns a number, called an ordinal element index, that is used by later records to refer to an ordinal element described by this record.

The name of the element is indicated by a literal index. Each ordinal element record contains the file index and source line number for the file and line in which the ordinal element was declared.

Additionally, each ordinal element record contains the following data:

- The ordinal type index of the ordinal type to which it belongs
- The value for that element

```
declare                              /*                            */
  1 xin_Ord_Elem   based( null() ), /*                            */
                                     /*                            */
    2 xin_Ord_Elem_Hdr               /* standard header            */
          like xin_Hdr,              /*                            */
                                     /*                            */
    2 xin_Ord_Elem_File_Id           /* file id                    */
          fixed bin(31) unsigned,    /*                            */
                                     /*                            */
    2 xin_Ord_Elem_Line_No           /* line no within file        */
          fixed bin(31) unsigned,    /*                            */
                                     /*                            */
    2 xin_Ord_Elem_Id                /* identifying number         */
                  fixed bin(31),     /*                            */
                                     /*                            */
    2 xin_Ord_Elem_Type_Id           /* id of ordinal type         */
                  fixed bin(31),     /*                            */
                                     /*                            */
    2 xin_Ord_Elem_Value             /* ordinal value              */
                  fixed bin(31),     /*                            */
                                     /*                            */
    2 xin_Ord_Elem_Name              /* ordinal name               */
                  fixed bin(31);     /*                            */
```

Figure 121. Declare for an ordinal element record

Symbol records

The declaration for the symbol record is in the member `ibmwxin` in the samples data set SIBMZSAM. Each symbol record assigns a number that is called a symbol index. The index is used by later records to refer to the symbol described by this record.

For example, the index can be used as the name of a user variable or constant. The name of the identifier is indicated by a literal index. Each symbol record contains the file index and source line number for the file and line in which the symbol was declared.

If the identifier is part of a structure or union, the symbol record contains a symbol index for each of the following:

- The first sibling, if any
- The parent, if any
- The first child, if any

Consider the following structure:

```
dcl
  1 a
    , 3 b     fixed bin
    , 3 c     fixed bin
    , 3 d
      , 5 e   fixed bin
      , 5 f   fixed bin
;
```

The symbol indices assigned to the elements of the preceding structure would be as follows:

```
symbol  index     sibling      parent    child
------  -----     -------      ------    -----
  a       1         0            0         2
  b       2         3            1         0
  c       3         4            1         0
  d       4         0            1         5
  e       5         6            4         0
  f       6         0            4         0
```

Figure 122. Symbol indices assigned to the elements of a structure

Each symbol record also contains a series of bit(1) fields that indicate if various attributes apply to this variable.

Each symbol record also contains the following elements:

User-given structure level
This is a user-given structure level for the identifier. For the element c of the structure above, the value is 3. For non-structure members, the value is set to 1.

Logican structure level
The logical structure level for the identifier For the element c of the structure above, the value is 2. For non-structure members, the value is set to 1.

Dimensions

The number of dimensions declared for the variable not counting any inherited dimensions.

The number of dimensions for the variable including all inherited dimensions.

Offset
The offset into the outermost parent structure.

Elemental size
Elemental size is in bytes unless the variable is bit aligned, in which case it is in bits. In either case, this does not factor any in dimensions.

Size
Size in bytes with its dimensions factored in.

Alignment
Identified by the following:

- 0 for bit-aligned
- 7 for byte-aligned
- 15 for halfword-aligned
- 31 for fullword-aligned
- 63 for quadword-aligned

A union within the record is dedicated to describing information that is dependent on the variable's storage class:

Static variables
If the variable is declared as external with a separate external name (dcl x ext('y')), the literal index of that name is specified.

Based variables
If the variable is declared as based on another mapped variable that is not an element of an array, the symbol index of that variable is specified.

Defined variables
If the variable is declared as defined on another mapped variable that is not an element of an array, the symbol index of that variable is specified here. If its position attribute is constant, it is also specified.

The variable's data type is specified by the ordinal shown in Figure 123.

```
define
  ordinal
    xin_Data_Kind
      (  xin_Data_Kind_Unset
        ,xin_Data_Kind_Character
        ,xin_Data_Kind_Bit
        ,xin_Data_Kind_Graphic
        ,xin_Data_Kind_Fixed
        ,xin_Data_Kind_Float
        ,xin_Data_Kind_Picture
        ,xin_Data_Kind_Pointer
        ,xin_Data_Kind_Offset
        ,xin_Data_Kind_Entry
        ,xin_Data_Kind_File
        ,xin_Data_Kind_Label
        ,xin_Data_Kind_Format
        ,xin_Data_Kind_Area
        ,xin_Data_Kind_Task
        ,xin_Data_Kind_Event
        ,xin_Data_Kind_Condition
        ,xin_Data_Kind_Structure
        ,xin_Data_Kind_Union
        ,xin_Data_Kind_Descriptor
        ,xin_Data_Kind_Ordinal
        ,xin_Data_Kind_Handle
        ,xin_Data_Kind_Type
        ,xin_Data_Kind_Builtin
        ,xin_Data_Kind_Generic
        ,xin_Data_Kind_Widechar
                                  )  prec(8) unsigned;
```

Figure 123. Data type of a variable

A union within the record is dedicated to describing information that is dependent on the variable's data type. Most of this information is self-explanatory (for example, the precision for an arithmetic type) except perhaps for the following variables:

Picture variables
The literal index of the picture specification is specified.

Entry variables
If the variable has the returns attribute, the symbol index of the returns description is specified.

Ordinal variables
The ordinal type index is specified.

Typed variables and handles
The symbol index of the underlying type is specified.

String and area variables
The type and value of the extent is specified in addition to the symbol index of the returns description. The type of the extent is encoded by the values:

```
declare
  (  xin_Extent_Constant     value(01)
    ,xin_Extent_Star         value(02)
    ,xin_Extent_Nonconstant  value(04)
    ,xin_Extent_Refer        value(08)
    ,xin_Extent_In_Error     value(16)
  )
  fixed bin;
```

If the element has any dimensions, the type and values for its lower and upper bounds are specified at the very end of the record. These fields are not present if the element has no dimensions.

Note that the attributes flags reflect the attributes after the compiler has applied all defaults. So, for example, every numeric variable (including numeric PICTURE variables) has either the REAL or COMPLEX attribute flag set.

Understanding SYSADATA syntax information

When you specify the SYN suboption of the XINFO compile-time option, the compiler generates a SYSADATA file that contains syntax information in addition to the records generated for the MSG and SYM suboptions.

The following records contain syntax information:

- Source records
- Token records
- Syntax records

Source records

Each source record assigns a number, called a source id, that is used by later records to refer to the source line described by this record. The line might be from the primary PL/I source file or an INCLUDE file, as indicated by the source file id and line number fields in the record. The rest of the record holds the actual data in the source line.

```
Dcl
  1 Xin_Src     Based( null() ),   /* source record                */
                                   /*                              */
    2 Xin_Src_Hdr                  /* standard header              */
           like Xin_Hdr,           /*                              */
                                   /*                              */
    2 Xin_Src_File_Id              /* file id                      */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Src_Line_No              /* line no within file          */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Src_Id                   /* id for this source record    */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Src_Length               /* length of text               */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Src_Text                 /* actual text                  */
        char( 137 refer(xin_Src_Length) );
```

Figure 124. Declare for a source record

Token records

Each token record assigns a number, called a token index, that is used by later records to refer to a token recognized by the PL/I compiler. The record also identifies the type of the token plus the column and line on which it started and ended.

```
Dcl
  1 Xin_Tok     Based( null() ),   /* token record                 */
                                   /*                              */
    2 Xin_Tok_Hdr                  /* standard header              */
           like Xin_Hdr,           /*                              */
                                   /*                              */
    2 Xin_Tok_Inx                  /* adata index for token        */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_Begin_Line           /* starting line no within file */
           fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_End_Line_Offset      /* offset of end line from first */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_Kind_Value           /* token kind                   */
           ordinal xin_Tok_Kind,   /*                              */
                                   /*                              */
    2 Xin_Tok_Rsrvd                /* reserved                     */
           fixed bin(8) unsigned,  /*                              */
                                   /*                              */
    2 Xin_Tok_Begin_Col            /* starting column              */
           fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Tok_End_Col              /* ending column                */
           fixed bin(16) unsigned; /*                              */
```

Figure 125. Declare for a token record

The ordinal xin_Tok_Kind identifies the type of the token record.

```
Define
 ordinal
   xin_Tok_Kind
     (  xin_Tok_Kind_Unset
       ,xin_Tok_Kind_Lexeme
       ,xin_Tok_Kind_Comment
       ,xin_Tok_Kind_Literal
       ,xin_Tok_Kind_Identifier
       ,xin_Tok_Kind_Keyword
                                  )  prec(8) unsigned;
```

Figure 126. Declare for the token record kind

Syntax records

Each syntax record assigns a number, called a node id, that is used by later records to refer to other syntax records. The first syntax record will have kind xin_Syn_Kind_Package, and if the compilation unit has any procedures, the child node of this record will point to the first of these procedures. The parent, sibling and child nodes will then provide a map with the appropriate relationships of all the procedures and begin blocks in the compilation unit.

Consider the following simple program:

```
a: proc;
  call b;
  call c;
  b: proc;
  end b;
  c: proc;
    call d;
    d: proc;
    end d;
  end c;
end a;
```

The node indices are assigned to the blocks of the preceding program as follows:

```
symbol  index     sibling      parent    child
------  -----     -------      ------    -----
  -       1          0           0         2
  a       2          0           1         3
  b       3          4           2         0
  c       4          0           2         5
  d       5          0           4         0
```

Figure 127. Node indices assigned to the blocks in a program

```
Dcl
  1 Xin_Syn     Based( null() ),   /* syntax record                */
                                   /*                              */
    2 Xin_Syn_Hdr                  /* standard header              */
          like Xin_Hdr,            /*                              */
                                   /*                              */
    2 Xin_Syn_Node_Id              /* node id                      */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Node_Kind            /* node type                    */
          ordinal xin_syn_kind,    /*                              */
                                   /*                              */
    2 Xin_Syn_Node_Exp_Kind        /* node sub type                */
          ordinal xin_exp_kind,    /*                              */
                                   /*                              */
    2 *                            /* reserved                     */
          fixed bin(16) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Parent_Node_Id       /* node id of parent            */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Sibling_Node_Id      /* node id of sibling           */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 Xin_Syn_Child_Node_Id        /* node id of child             */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 xin_Syn_First_Tok            /* id of first spanned token    */
          fixed bin(32) unsigned, /*                              */
                                   /*                              */
    2 xin_Syn_Last_Tok             /* id of last spanned token     */
          fixed bin(32) unsigned, /*                              */
```

Figure 128. Declare for a syntax record

```
                                    /*                              */
  2 * union,                     /* qualifier for node            */
                                 /*                               */
    3 Xin_Syn_Int_Value          /* used if int                   */
         fixed bin(31),          /*                               */
                                 /*                               */
    3 Xin_Syn_Literal_Id         /* used if name, number, picture */
         fixed bin(31),          /*                               */
                                 /*                               */
    3 Xin_Syn_Node_Lex           /* used if lexeme, assignment,   */
         ordinal xin_Lex_kind,   /* infix_op, prefix_op           */
                                 /*                               */
    3 Xin_Syn_Node_Voc           /* used if keyword, end_for_do   */
         ordinal xin_Voc_kind,   /*                               */
                                 /*                               */
    3 Xin_Syn_Block_Node         /* used if call_begin            */
         fixed bin(31),          /*   to hold node of begin block */
                                 /*                               */
    3 Xin_Syn_Bif_Id             /* used if bif_rfrnc             */
         fixed bin(32) unsigned, /*                               */
                                 /*                               */
    3 Xin_Syn_Sym_Id             /* used if label, unsub_rfrnc,   */
         fixed bin(32) unsigned, /* subscripted_rfrnc             */
                                 /*                               */
    3 Xin_Syn_Proc_Data,         /* used if package, proc or begin*/
                                 /*                               */
      4 Xin_Syn_First_Sym        /*   id of first contained sym   */
         fixed bin(32) unsigned, /*                               */
                                 /*                               */
      4 Xin_Syn_Block_Sym        /*   id of sym for this block    */
         fixed bin(32) unsigned, /*                               */
```

Declare for a syntax record (continued)

```
                                  /*                              */
    3 Xin_Syn_Number_Data,        /* used if number                */
                                  /*                               */
      4 Xin_Syn_Number_Id         /*   id of literal               */
        fixed bin(32) unsigned,   /*                               */
                                  /*                               */
      4 Xin_Syn_Number_Type       /*   type                        */
        ordinal xin_Number_Kind,  /*                               */
                                  /*                               */
      4 Xin_Syn_Number_Prec       /*   precision                   */
        fixed bin(8) unsigned,    /*                               */
                                  /*                               */
      4 Xin_Syn_Number_Scale      /*   scale factor                */
        fixed bin(7) signed,      /*                               */
                                  /*                               */
      4 Xin_Syn_Number_Bytes      /*   bytes it would occupy       */
        fixed bin(8) unsigned,    /*   in its internal form        */
                                  /*                               */
    3 Xin_Syn_String_Data,        /* used if char_string,          */
                                  /* bit_string, graphic_string    */
                                  /*                               */
      4 Xin_Syn_String_Id         /*   id of literal               */
        fixed bin(32) unsigned,   /*                               */
                                  /*                               */
      4 Xin_Syn_String_Len        /*   string length in its units  */
        fixed bin(32) unsigned,   /*                               */
                                  /*                               */
    3 Xin_Syn_Stmt_Data,          /* used if stmt                  */
                                  /*                               */
      4 Xin_Syn_File_Id           /*   file id                     */
        fixed bin(32) unsigned,   /*                               */
                                  /*                               */
      4 Xin_Syn_Line_No           /*   line no within file         */
        fixed bin(32) unsigned,   /*                               */
                                  /*                               */
  2 *                 char(0);    /*                               */
```

Declare for a syntax record (continued)

The ordinal xin_Syn_Kind identifies the type of the syntax record.

```
Define
  ordinal
    xin_Syn_Kind
      (  xin_Syn_Kind_Unset
       ,xin_Syn_Kind_Lexeme
       ,xin_Syn_Kind_Asterisk
       ,xin_Syn_Kind_Int
       ,xin_Syn_Kind_Name
       ,xin_Syn_Kind_Expression
       ,xin_Syn_Kind_Parenthesized_Expr
       ,xin_Syn_Kind_Argument_List
       ,xin_Syn_Kind_Keyword
       ,xin_Syn_Kind_Proc_Stmt
       ,xin_Syn_Kind_Begin_Stmt
       ,xin_Syn_Kind_Stmt
       ,xin_Syn_Kind_Substmt
       ,xin_Syn_Kind_Label
       ,xin_Syn_Kind_Invoke_Begin
       ,xin_Syn_Kind_Assignment
       ,xin_Syn_Kind_Assignment_Byname
       ,xin_Syn_Kind_Do_Fragment
       ,xin_Syn_Kind_Keyed_List
       ,xin_Syn_Kind_Iteration_Factor
       ,xin_Syn_Kind_If_Clause
       ,xin_Syn_Kind_Else_Clause
       ,xin_Syn_Kind_Do_Stmt
       ,xin_Syn_Kind_Select_Stmt
       ,xin_Syn_Kind_When_Stmt
       ,xin_Syn_Kind_Otherwise_Stmt
       ,xin_Syn_Kind_Procedure
       ,xin_Syn_Kind_Package
       ,xin_Syn_Kind_Begin_Block
       ,xin_Syn_Kind_Picture
       ,xin_Syn_Kind_Raw_Rfrnc
       ,xin_Syn_Kind_Generic_Desc
                                  )  prec(8) unsigned;
```

Figure 129. Declare for the syntax record kind

Consider the following simple program:

```
a: proc(x);
  dcl x char(8);
  x = substr(datetime(),1,8);
end;
```

The node indices are assigned to the blocks of the preceding program as follows:

node_kind	index	sibling	parent	child
package	1	0	0	2
procedure	2	0	1	0
expression	3	0	0	0
stmt	4	5	2	6
stmt	5	10	2	11
label	6	7	4	0
keyword	7	8	4	0
expression	8	9	4	0
lexeme	9	0	4	0
stmt	10	0	2	18
assignment	11	12	5	13
lexeme	12	0	5	0
expression	13	14	11	0
expression	14	0	11	15
expression	15	16	14	0
expression	16	17	14	0
expression	17	0	14	0
keyword	18	19	10	0
lexeme	19	0	10	0

Figure 130. Node indices assigned to the syntax records in a program

The procedure record contains the identifier (in the block_sym field) for the symbol record for ENTRY A. This symbol record contains, in turn, the node identifier (in the first_stmt_id field) for the first statement in that procedure.

Note that for the statement records, the sibling node identifier points to the next statement record, if any; the child node identifier points to the first element of that statement record.

The records for the PROCEDURE statement consists of 4 records:

- A label record
- A keyword record (for the PROCEDURE keyword)
- An expression record (for the parameter X) with expression kind of unsub_rfrnc and a sym_id for the symbol X
- A lexeme record (for the semicolon)

The records for the assignment statement consists of 2 records:

- An assignment record that has 2 children:
  - An expression record (for the target X) with expression kind of unsub_rfrnc and a sym_id for the symbol X
  - An expression record (for the source) with expression kind of builtin_rfrnc and a sym_id for the symbol SUBSTR, and this record has itself 3 children:
    - An expression record (for the first argument) with expression kind of builtin_rfrnc and a sym_id for the symbol DATETIME
    - An expression record (for the second argument) with expression kind of number and a literal_id for the value 1
    - An expression record (for the third argument) with expression kind of number and a literal_id for the value 8
- A lexeme record (for the semicolon)

The records for the END statement consists of 2 records:

- A keyword record (for the END keyword)

- A lexeme record (for the semicolon)

The ordinal xin_Exp_Kind identifies the type of an expression for a syntax record that describes an expression. Some of these records will have nonzero child nodes; for example:

- An infix op, such as a minus for a subtraction, will have a child node that describes its lefthand operand (and the sibling node of that operand will describe the righthand operator).
- A prefix op, such as a minus for a negation, will have a child node that describes its operand.

```
Define
 ordinal
   xin_Exp_Kind
     (  xin_Exp_Kind_Unset
       ,xin_Exp_Kind_Bit_String
       ,xin_Exp_Kind_Char_String
       ,xin_Exp_Kind_Graphic_String
       ,xin_Exp_Kind_Number
       ,xin_Exp_Kind_Infix_Op
       ,xin_Exp_Kind_Prefix_Op
       ,xin_Exp_Kind_Builtin_Rfrnc
       ,xin_Exp_Kind_Entry_Rfrnc
       ,xin_Exp_Kind_Qualified_Rfrnc
       ,xin_Exp_Kind_Unsub_Rfrnc
       ,xin_Exp_Kind_Subscripted_Rfrnc
       ,xin_Exp_Kind_Type_Func
       ,xin_Exp_Kind_Widechar_String
                                  )  prec(8) unsigned;
```

Figure 131. Declare for the expression kind

The ordinal xin_Number_Kind identifies the type of a number for a syntax record that describes a number.

```
Define
 ordinal
   xin_Number_Kind
     (  xin_Number_Kind_Unset
       ,xin_Number_Kind_Real_Fixed_Bin
       ,xin_Number_Kind_Real_Fixed_Dec
       ,xin_Number_Kind_Real_Float_Bin
       ,xin_Number_Kind_Real_Float_Dec
       ,xin_Number_Kind_Cplx_Fixed_Bin
       ,xin_Number_Kind_Cplx_Fixed_Dec
       ,xin_Number_Kind_Cplx_Float_Bin
       ,xin_Number_Kind_Cplx_Float_Dec
                                  )  prec(8) unsigned;
```

Figure 132. Declare for the number kind

The ordinal xin_Lex_Kind identifies the type of a lexeme for a syntax record that describes a lexical unit.

- In these ordinal names, "vrule" means "vertical rule", which is used, for instance, as the "or" symbol.

- In these ordinal names, "dbl" means "double", so that dbl_Vrule is a doubled vertical rule that is used, for instance, as the "concatenate" symbol.

```
Define
 ordinal
   xin_Lex_Kind
     (  xin_Lex_Undefined
       ,xin_Lex_Period
       ,xin_Lex_Colon
       ,xin_Lex_Semicolon
       ,xin_Lex_Lparen
       ,xin_Lex_Rparen
       ,xin_Lex_Comma
       ,xin_Lex_Equals
       ,xin_Lex_Gt
       ,xin_Lex_Ge
       ,xin_Lex_Lt
       ,xin_Lex_Le
       ,xin_Lex_Ne
       ,xin_Lex_Lctr
       ,xin_Lex_Star
       ,xin_Lex_Dbl_Colon
       ,xin_Lex_Not
       ,xin_Lex_Vrule
       ,xin_Lex_Dbl_Vrule
       ,xin_Lex_And
       ,xin_Lex_Dbl_Star
       ,xin_Lex_Plus
       ,xin_Lex_Minus
       ,xin_Lex_Slash
       ,xin_Lex_Equals_Gt
       ,xin_Lex_Lparen_Colon
       ,xin_Lex_Colon_Rparen
       ,xin_Lex_Plus_Equals
       ,xin_Lex_Minus_Equals
       ,xin_Lex_Star_Equals
       ,xin_Lex_Slash_Equals
       ,xin_Lex_Vrule_Equals
       ,xin_Lex_And_Equals
       ,xin_Lex_Dbl_Star_Equals
       ,xin_Lex_Dbl_Vrule_Equals
       ,xin_Lex_Dbl_Slash
     ) unsigned prec(16);
```

Figure 133. Declare for the lexeme kind

The ordinal xin_Voc_Kind identifies the keyword for a syntax record that describes an item from the compiler's "vocabulary".

The declaration for the voc kind is provided in the member `ibmwxin` in the samples data set SIBMZSAM.

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 Corporation
J74/G4
555 Bailey Avenue
San Jose, CA 95141-1099
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

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 publication 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.

© Copyright IBM Corp. 1999, 2017

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Intel is a registered trademark of Intel Corporation in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and other countries.

Pentium is a registered trademark of Intel Corporation in the United States and other countries.

Unicode is a trademark of the Unicode Consortium.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be the trademarks or service marks of others.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Bibliography

PL/I publications

Enterprise PL/I for z/OS

Programming Guide, GI11-9145
Language Reference, SC14-7285
Messages and Codes, GC14-7286
Compiler and Run-Time Migration Guide, GC14-7284

PL/I for MVS & VM

Installation and Customization under MVS, SC26-3119
Language Reference, SC26-3114
Compile-Time Messages and Codes, SC26-3229
Diagnosis Guide, SC26-3149
Migration Guide, SC26-3118
Programming Guide, SC26-3113
Reference Summary, SX26-3821

Enterprise PL/I for AIX

Programming Guide, SC14-7319
Language Reference, SC14-7320
Messages and Codes, GC14-7321
Installation Guide, GC14-7322

Related publications

DB2 for z/OS

Administration Guide, SC19-2968

Application Programming and SQL Guide, SC19-2969

Command Reference, SC19-2972

Messages, GC19-2979

Codes, GC19-2971

SQL Reference, SC19-2983

See also the Information Center: publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db2z10.doc/src/alltoc/db2z_10_prodhome.htm

LOBs with DB2 for z/OS: Stronger and Faster, SG24-7270

DFSORT

Application Programming Guide, SC33-4035

Installation and Customization, SC33-4034

IMS/ESA®

Application Programming: Database Manager, SC26-8015

Application Programming: Database Manager Summary, SC26-8037

Application Programming: Design Guide, SC26-8016

Application Programming: Transaction Manager, SC26-8017

Application Programming: Transaction Manager Summary, SC26-8038

Application Programming: EXEC DL/I Commands for CICS and IMS™, SC26-8018

© Copyright IBM Corp. 1999, 2017

Application Programming: EXEC DL/I Commands for CICS and IMS Summary, SC26-8036

TXSeries® for Multiplatforms

Encina Administration Guide Volume 2: Server Administration, SC09-4474

Encina SFS Programming Guide, SC09-4483

See also the Information Center: publib.boulder.ibm.com/infocenter/txformp/v7r1/index.jsp

z/Architecture

Principles of Operation, SA22-7832

z/OS Language Environment

Concepts Guide, SA22-7567

Debugging Guide, GA22-7560

Run-Time Messages, SA22-7566

Customization, SA22-7564

Programming Guide, SA22-7561

Programming Reference, SA22-7562

Run-Time Application Migration Guide, GA22-7565

Writing Interlanguage Communication Applications, SA22-7563

z/OS MVS

JCL Reference, SA22-7597

JCL User's Guide, SA22-7598

System Commands, SA22-7627

z/OS TSO/E

Command Reference, SA22-7782

User's Guide, SA22-7794

z/OS UNIX System Services

z/OS UNIX System Services Command Reference, SA22-7802

z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA22-7803

z/OS UNIX System Services User's Guide, SA22-7801

Unicode and character representation

z/OS Support for Unicode: Using Conversion Services, SC33-7050

Glossary

This glossary defines terms for all platforms and releases of PL/I. It might contain terms that this manual does not use. If you do not find the terms for which you are looking, see the index in this manual or *IBM Dictionary of Computing*, SC20-1699.

A

access To reference or retrieve data.

action specification
In an ON statement, the ON-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever the appropriate condition is raised.

activate (a block)
To initiate the execution of a block. A procedure block is activated when it is invoked. A begin-block is activated when it is encountered in the normal flow of control, including a branch. A package cannot be activated.

activate (a preprocessor variable or preprocessor entry point)
To make a macro facility identifier eligible for replacement in subsequent source code. The %ACTIVATE statement activates preprocessor variables or preprocessor entry points.

active The state of a block after activation and before termination. The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text. The state in which an event variable is said to be during the time it is associated with an asynchronous operation. The state in which a task variable is said to be when its associated task is attached. The state in which a task is said to be before it has been terminated.

actual origin (AO)
The location of the first item in the array or structure.

additive attribute
A file description attribute for which there are no defaults, and which, if required, must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute*.

adjustable extent
The bound (of an array), the length (of a string), or the size (of an area) that might be different for different generations of the associated variable. Adjustable extents are specified as expressions or asterisks (or by REFER options for based variables), which are evaluated separately for each generation. They cannot be used for static variables.

aggregate
See *data aggregate*.

aggregate expression
An array, structure, or union expression.

aggregate type
For any item of data, the specification whether it is structure, union, or array.

allocated variable
A variable with which main storage is associated and not freed.

allocation
The reservation of main storage for a variable. A generation of an allocated variable. The association of a PL/I file with a system data set, device, or file.

alignment
The storing of data items in relation to certain machine-dependent boundaries (for example, a fullword or halfword boundary).

alphabetic character
Any of the characters A through Z of the English alphabet and the alphabetic extenders #, $, and @ (which can have a different graphic representation in different countries).

alphameric character
An alphabetic character or a digit.

alternative attribute
A file description attribute that is chosen from a group of attributes. If none is specified, a default is assumed. Contrast with *additive attribute*.

© Copyright IBM Corp. 1999, 2017

ambiguous reference
A reference that is not sufficiently qualified to identify one and only one name known at the point of reference.

area A portion of storage within which based variables can be allocated.

argument
An expression in an argument list as part of an invocation of a subroutine or function.

argument list
A parenthesized list of zero or more arguments, separated by commas, following an entry name constant, an entry name variable, a generic name, or a built-in function name. The list becomes the parameter list of the entry point.

arithmetic comparison
A comparison of numeric values. See also *bit comparison*, *character comparison*.

arithmetic constant
A fixed-point constant or a floating-point constant. Although most arithmetic constants can be signed, the sign is not part of the constant.

arithmetic conversion
The transformation of a value from one arithmetic representation to another.

arithmetic data
Data that has the characteristics of base, scale, mode, and precision. Coded arithmetic data and pictured numeric character data are included.

arithmetic operators
Either of the prefix operators + and -, or any of the following infix operators: + - * / **

array A named, ordered collection of one or more data elements with identical attributes, grouped into one or more dimensions.

array expression
An expression whose evaluation yields an array of values.

array of structures
An ordered collection of identical structures specified by giving the dimension attribute to a structure name.

array variable
A variable that represents an aggregate of data items that must have identical attributes. Contrast with *structure variable*.

ASCII American National Standard Code for Information Interchange.

assignment
The process of giving a value to a variable.

asynchronous operation
The overlap of an input/output operation with the execution of statements. The concurrent execution of procedures using multiple flows of control for different tasks.

attachment of a task
The invocation of a procedure and the establishment of a separate flow of control to execute the invoked procedure (and procedures it invokes) asynchronously, with execution of the invoking procedure.

attention
An occurrence, external to a task, that could cause a task to be interrupted.

attribute
A descriptive property associated with a name to describe a characteristic represented. A descriptive property used to describe a characteristic of the result of evaluation of an expression.

automatic storage allocation
The allocation of storage for automatic variables.

automatic variable
A variable whose storage is allocated automatically at the activation of a block and released automatically at the termination of that block.

B

base The number system in which an arithmetic value is represented.

base element
A member of a structure or a union that is itself not another structure or union.

base item
The automatic, controlled, or static variable or the parameter upon which a defined variable is defined.

based reference
A reference that has the based storage class.

based storage allocation
The allocation of storage for based variables.

based variable
A variable whose storage address is provided by a locator. Multiple generations of the same variable are accessible. It does not identify a fixed location in storage.

begin-block
A collection of statements delimited by BEGIN and END statements, forming a name scope. A begin-block is activated either by the raising of a condition (if the begin-block is the action specification for an ON-unit) or through the normal flow of control, including any branch resulting from a GOTO statement.

binary A number system whose only numerals are 0 and 1.

binary digit
See *bit*.

binary fixed-point value
An integer consisting of binary digits and having an optional binary point and optional sign. Contrast with *decimal fixed-point value*.

binary floating-point value
An approximation of a real number in the form of a significand, which can be considered as a binary fraction, and an exponent, which can be considered as an integer exponent to the base of 2. Contrast with *decimal floating-point value*.

bit A 0 or a 1. The smallest amount of space of computer storage.

bit comparison
A left-to-right, bit-by-bit comparison of binary digits. See also *arithmetic comparison, character comparison*.

bit string constant
A series of binary digits enclosed in and followed immediately by the suffix B. Contrast with *character constant*. A series of hexadecimal digits enclosed in single quotes and followed by the suffix B4.

bit string
A string composed of zero or more bits.

bit string operators
The logical operators not and exclusive-or (¬), and (&), and or (|).

bit value
A value that represents a bit type.

block A sequence of statements, processed as a unit, that specifies the scope of names and the allocation of storage for names declared within it. A block can be a package, procedure, or a begin-block.

bounds
The upper and lower limits of an array dimension.

break character
The underscore symbol (_). It can be used to improve the readability of identifiers. For instance, a variable could be called OLD_INVENTORY_TOTAL instead of OLDINVENTORYTOTAL.

built-in function
A predefined function supplied by the language, such as SQRT (square root).

built-in function reference
A built-in function name, which has an optional argument list.

built-in name
The entry name of a built-in subroutine.

built-in subroutine
Subroutine that has an entry name that is defined at compile-time and is invoked by a CALL statement.

buffer Intermediate storage, used in input/output operations, into which a record is read during input and from which a record is written during output.

C

call To invoke a subroutine by using the CALL statement or CALL option.

character comparison
A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison, bit comparison*.

character string constant
A sequence of characters enclosed in single quotes; for example, 'Shakespeare''s 'Hamlet:''.

character set
A defined collection of characters. See also *ASCII* and *EBCDIC*.

character string picture data
Picture data that has only a character value. This type of picture data must have at least one A or X picture specification character. Contrast with *numeric picture data*.

closing (of a file)
The dissociation of a file from a data set or device.

coded arithmetic data
Data items that represent numeric values and are characterized by their base (decimal or binary), scale (fixed-point or floating-point), and precision (the number of digits each can have). This data is stored in a form that is acceptable, without conversion, for arithmetic calculations.

combined nesting depth
The deepest level of nesting, determined by counting the levels of PROCEDURE/BEGIN/ON, DO, SELECT, and IF...THEN...ELSE nestings in the program.

comment
A string of zero or more characters used for documentation that are delimited by /* and */.

commercial character

- CR (credit) picture specification character
- DB (debit) picture specification character

comparison operator
An operator that can be used in an arithmetic, string locator, or logical relation to indicate the comparison to be done between the terms in the relation. The comparison operators are:

= (equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

¬= (not equal to)

¬> (not greater than)

¬< (not less than)

compile time
In general, the time during which a source program is translated into an object module. In PL/I, it is the time during which a source program can be altered, if desired, and then translated into an object program.

compiler options
Keywords that are specified to control certain aspects of a compilation, such as: the nature of the object module generated, the types of printed output produced, and so forth.

complex data
Arithmetic data, each item of which consists of a real part and an imaginary part.

composite operator
An operator that consists of more than one special character, such as <=, **, and /*.

compound statement
A statement that contains other statements. In PL/I, IF, ON, OTHERWISE, and WHEN are the only compound statements. See *statement body*.

concatenation
The operation that joins two strings in the order specified, forming one string whose length is equal to the sum of the lengths of the two original strings. It is specified by the operator ||.

condition
An exceptional situation, either an error (such as an overflow), or an expected situation (such as the end of an input file). When a condition is raised (detected), the action established for it is processed. See also *established action* and *implicit action*.

condition name
Name of a PL/I-defined or programmer-defined condition.

condition prefix
A parenthesized list of one or more condition names prefixed to a statement.

It specifies whether the named conditions are to be enabled or disabled.

connected aggregate
An array or structure whose elements occupy contiguous storage without any intervening data items. Contrast with *nonconnected aggregate*.

connected reference
A reference to connected storage. It must be apparent, prior to execution of the program, that the storage is connected.

connected storage
Main storage of an uninterrupted linear sequence of items that can be referred to by a single name.

constant
An arithmetic or string data item that does not have a name and whose value cannot change. An identifier declared with the VALUE attribute. An identifier declared with the FILE or the ENTRY attribute but without the VARIABLE attribute.

constant reference
A value reference which has a constant as its object

contained block, declaration, or source text
All blocks, procedures, statements, declarations, or source text inside a begin, procedure, or a package block. The entire package, procedure, and the BEGIN statement and its corresponding END statements are not contained in the block.

containing block
The package, procedure, or begin-block that contains the declaration, statement, procedure, or other source text in question.

contextual declaration
The appearance of an identifier that has not been explicitly declared in a DECLARE statement, but whose context of use allows the association of specific attributes with the identifier.

control character
A character in a character set whose occurrence in a particular context specifies a control function. One example is the end-of-file (EOF) marker.

control format item
A specification used in edit-directed transmission to specify positioning of a data item within the stream or printed page.

control variable
A variable that is used to control the iterative execution of a DO statement.

controlled parameter
A parameter for which the CONTROLLED attribute is specified in a DECLARE statement. It can be associated only with arguments that have the CONTROLLED attribute.

controlled storage allocation
The allocation of storage for controlled variables.

controlled variable
A variable whose allocation and release are controlled by the ALLOCATE and FREE statements, with access to the current generation only.

control sections
Grouped machine instructions in an object module.

conversion
The transformation of a value from one representation to another to conform to a given set of attributes. For example, converting a character string to an arithmetic value such as FIXED BINARY (15,0).

cross section of an array
The elements represented by the extent of at least one dimension of an array. An asterisk in the place of a subscript in an array reference indicates the entire extent of that dimension.

current generation
The generation of an automatic or controlled variable that is currently available by referring to the name of the variable.

D

data Representation of information or of value in a form suitable for processing.

data aggregate
A data item that is a collection of other data items.

data attribute
A keyword that specifies the type of data that the data item represents, such as FIXED BINARY.

data-directed transmission
The type of stream-oriented transmission in which data is transmitted. It resembles an assignment statement and is of the form `name = constant`.

data item
A single named unit of data.

data list
In stream-oriented transmission, a parenthesized list of the data items used in GET and PUT statements. Contrast with *format list*.

data set
A collection of data external to the program that can be accessed by reference to a single file name. A device that can be referenced.

data specification
The portion of a stream-oriented transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list(s) and, for edit-directed mode, the format list(s).

data stream
Data being transferred from or to a data set by stream-oriented transmission, as a continuous stream of data elements in character form.

data transmission
The transfer of data from a data set to the program or vice versa.

data type
A set of data attributes.

DBCS In the character set, each character is represented by two consecutive bytes.

deactivated
The state in which an identifier is said to be when its value cannot replace a preprocessor identifier in source program text. Contrast with *active*.

debugging
Process of removing bugs from a program.

decimal
The number system whose numerals are 0 through 9.

decimal digit picture character
The picture specification character 9.

decimal fixed-point constant
A constant consisting of one or more decimal digits with an optional decimal point.

decimal fixed-point value
A rational number consisting of a sequence of decimal digits with an assumed position of the decimal point. Contrast with *binary fixed-point value*.

decimal floating-point constant
A value made up of a significand that consists of a decimal fixed-point constant, and an exponent that consists of the letter E followed by an optionally signed integer constant not exceeding three digits.

decimal floating-point value
An approximation of a real number, in the form of a significand, which can be considered as a decimal fraction, and an exponent, which can be considered as an integer exponent to the base 10. Contrast with *binary floating-point value*.

decimal picture data
See *numeric picture data.*

declaration
The establishment of an identifier as a name and the specification of a set of attributes (partial or complete) for it. A source of attributes of a particular name.

default
Describes a value, attribute, or option that is assumed when none has been specified.

defined variable
A variable that is associated with some or all of the storage of the designated base variable.

delimit
To enclose one or more items or statements with preceding and following characters or keywords.

delimiter
All comments and the following characters: percent, parentheses, comma, period, semicolon, colon, assignment symbol, blank, pointer, asterisk, and single quote. They define the limits of identifiers, constants, picture specifications, iSUBs, and keywords.

descriptor
A control block that holds information about a variable, such as area size, array bounds, or string length.

digit One of the characters 0 through 9.

dimension attribute
An attribute that specifies the number of dimensions of an array and indicates the bounds of each dimension.

disabled
The state of a condition in which no interrupt occurs and no established action will take place.

do-group
A sequence of statements delimited by a DO statement and ended by its corresponding END statement, used for control purposes. Contrast with *block*.

do-loop
See *iterative do-group*.

dummy argument
Temporary storage that is created automatically to hold the value of an argument that cannot be passed by reference.

dump Printout of all or part of the storage used by a program as well as other program information, such as a trace of an error's origin.

E

EBCDIC
(Extended Binary-Coded Decimal Interchange Code). A coded character set consisting of 8-bit coded characters.

edit-directed transmission
The type of stream-oriented transmission in which data appears as a continuous stream of characters and for which a format list is required to specify the editing desired for the associated data list.

element
A single item of data as opposed to a collection of data items such as an array; a scalar item.

element expression
An expression whose evaluation yields an element value.

element variable
A variable that represents an element; a scalar variable.

elementary name
See *base element*.

enabled
The state of a condition in which the condition can cause an interrupt and then invocation of the appropriate established ON-unit.

end-of-step message
message that follows the listng of the job control statements and job scheduler messages and contains return code indicating success or failure for each step.

entry constant
The label prefix of a PROCEDURE statement (an entry name). The declaration of a name with the ENTRY attribute but without the VARIABLE attribute.

entry data
A data item that represents an entry point to a procedure.

entry expression
An expression whose evaluation yields an entry name.

entry name
An identifier that is explicitly or contextually declared to have the ENTRY attribute (unless the VARIABLE attribute is given) or An identifier that has the value of an entry variable with the ENTRY attribute implied.

entry point
A point in a procedure at which it can be invoked. *primary entry point* and *secondary entry point*.

entry reference
An entry constant, an entry variable reference, or a function reference that returns an entry value.

entry variable
A variable to which an entry value can be assigned. It must have both the ENTRY and VARIABLE attributes.

entry value
The entry point represented by an entry constant or variable; the value includes

the environment of the activation that is associated with the entry constant.

environment (of an activation)
Information associated with and used in the invoked block regarding data declared in containing blocks.

environment (of a label constant)
Identity of the particular activation of a block to which a reference to a statement-label constant applies. This information is determined at the time a statement-label constant is passed as an argument or is assigned to a statement-label variable, and it is passed or assigned along with the constant.

established action
The action taken when a condition is raised. See also *implicit action* and *ON-statement action*.

epilogue
Those processes that occur automatically at the termination of a block or task.

evaluation
The reduction of an expression to a single value, an array of values, or a structured set of values.

event An activity in a program whose status and completion can be determined from an associated event variable.

event variable
A variable with the EVENT attribute that can be associated with an event. Its value indicates whether the action has been completed and the status of the completion.

explicit declaration
The appearance of an identifier (a name) in a DECLARE statement, as a label prefix, or in a parameter list. Contrast with *implicit declaration*.

exponent characters
The following picture specification characters:

1. K and E, which are used in floating-point picture specifications to indicate the beginning of the exponent field.
2. F, the scaling factor character, specified with an integer constant that indicates the number of decimal positions the decimal point is to be moved from its assumed position to the right (if the constant is positive) or to the left (if the constant is negative).

expression
A notation, within a program, that represents a value, an array of values, or a structured set of values. A constant or a reference appearing alone, or a combination of constants and/or references with operators.

extended alphabet
The uppercase and lowercase alphabetic characters A through Z, $, @ and #, or those specified in the NAMES compiler option.

extent The range indicated by the bounds of an array dimension, by the length of a string, or by the size of an area. The size of the target area if this area were to be assigned to a target area.

external name
A name (with the EXTERNAL attribute) whose scope is not necessarily confined only to one block and its contained blocks.

external procedure
A procedure that is not contained in any other procedure. A level-2 procedure contained in a package that is also exported.

external symbol
Name that can be referred to in a control section other than the one in which it is defined.

External Symbol Dictionary (ESD)
Table containing all the external symbols that appear in the object module.

extralingual character
Characters (such as $, @, and #) that are not classified as alphanumeric or special. This group includes characters that are determined with the NAMES compiler option.

F

factoring
The application of one or more attributes to a parenthesized list of names in a DECLARE statement, eliminating the repetition of identical attributes for multiple names.

field (in the data stream)
That portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

field (of a picture specification)
Any character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number.

file A named representation, within a program, of a data set or data sets. A file is associated with the data set(s) for each opening.

file constant
A name declared with the FILE attribute but not the VARIABLE attribute.

file description attributes
Keywords that describe the individual characteristics of each file constant. See also *alternative attribute* and *additive attribute*.

file expression
An expression whose evaluation yields a value of the type file.

file name
A name declared for a file.

file variable
A variable to which file constants can be assigned. It has the attributes FILE and VARIABLE and cannot have any of the file description attributes.

fixed-point constant
See *arithmetic constant*.

fix-up A solution, performed by the compiler after detecting an error during compilation, that allows the compiled program to run.

floating-point constant
See *arithmetic constant*.

flow of control
Sequence of execution.

format
A specification used in edit-directed data transmission to describe the representation of a data item in the stream (data format item) or the specific positioning of a data item within the stream (control format item).

format constant
The label prefix on a FORMAT statement.

format data
A variable with the FORMAT attribute.

format label
The label prefix on a FORMAT statement.

format list
In stream-oriented transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

fully qualified name
A name that includes all the names in the hierarchical sequence above the member to which the name refers, as well as the name of the member itself.

function (procedure)
A procedure that has a RETURNS option in the PROCEDURE statement. A name declared with the RETURNS attribute. It is invoked by the appearance of one of its entry names in a function reference and it returns a scalar value to the point of reference. Contrast with *subroutine*.

function reference
An entry constant or an entry variable, either of which must represent a function, followed by a possibly empty argument list. Contrast with *subroutine call*.

G

generation (of a variable)
The allocation of a static variable, a particular allocation of a controlled or automatic variable, or the storage indicated by a particular locator qualification of a based variable or by a defined variable or parameter.

generic descriptor
A descriptor used in a GENERIC attribute.

generic key
A character string that identifies a class of keys. All keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF', are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded

keys can be considered to be unique members of the classes 'ABCD', 'ABCE', 'ABDF', respectively.

generic name
The name of a family of entry names. A reference to the generic name is replaced by the entry name whose parameter descriptors match the attributes of the arguments in the argument list at the point of invocation.

group A collection of statements contained within larger program units. A group is either a do-group or a select-group and it can be used wherever a single statement can appear, except as an on-unit.

H

hex See *hexadecimal digit*.

hexadecimal
Pertaining to a numbering system with a base of sixteen; valid numbers use the digits 0 through 9 and the characters A through F, where A represents 10 and F represents 15.

hexadecimal digit
One of the digits 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively.

I

identifier
A string of characters, not contained in a comment or constant, and preceded and followed by a delimiter. The first character of the identifier must be one of the 26 alphabetic characters and extralingual characters, if any. The other characters, if any, can additionally include extended alphabetic, digit, or the break character.

IEEE Institute of Electrical and Electronics Engineers.

implicit
The action taken in the absence of an explicit specification.

implicit action
The action taken when an enabled condition is raised and no ON-unit is currently established for the condition. Contrast with *ON-statement action*.

implicit declaration
A name not explicitly declared in a DECLARE statement or contextually declared.

implicit opening
The opening of a file as the result of an input or output statement other than the OPEN statement.

infix operator
An operator that appears between two operands.

inherited dimensions
For a structure, union, or element, those dimensions that are derived from the containing structures. If the name is an element that is not an array, the dimensions consist entirely of its inherited dimensions. If the name is an element that is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure with one or more inherited dimensions is called a nonconnected aggregate. Contrast with *connected aggregate*.

input/output
The transfer of data between auxiliary medium and main storage.

insertion point character
A picture specification character that is, on assignment of the associated data to a character string, inserted in the indicated position. When used in a P-format item for input, the insertion character is used for checking purposes.

integer
An optionally signed sequence of digits or a sequence of bits without a decimal or binary point. An optionally signed whole number, commonly described as FIXED BINARY (p,0) or FIXED DECIMAL (p,0).

integral boundary
A byte multiple address of any 8-bit unit on which data can be aligned. It usually is a halfword, fullword, or doubleword (2-, 4-, or 8-byte multiple respectively) boundary.

interleaved array
An array that refers to nonconnected storage.

interleaved subscripts
Subscripts that exist in levels other than the lowest level of a subscripted qualified reference.

internal block
A block that is contained in another block.

internal name
A name that is known only within the block in which it is declared, and possibly within any contained blocks.

internal procedure
A procedure that is contained in another block. Contrast with *external procedure*.

interrupt
The redirection of the program's flow of control as the result of raising a condition or attention.

invocation
The activation of a procedure.

invoke
To activate a procedure.

invoked procedure
A procedure that has been activated.

invoking block
A block that activates a procedure.

iteration factor
In an INITIAL attribute specification, an expression that specifies the number of consecutive elements of an array that are to be initialized with the given value. In a format list, an expression that specifies the number of times a given format item or list of format items is to be used in succession.

iterative do-group
A do-group whose DO statement specifies a control variable and/or a WHILE or UNTIL option.

K

key Data that identifies a record within a direct access data set. See *source key* and *recorded key*.

keyword
An identifier that has a specific meaning in PL/I when used in a defined context.

keyword statement
A simple statement that begins with a keyword, indicating the function of the statement.

known (applied to a name)
Recognized with its declared meaning. A name is known throughout its scope.

L

label A name prefixed to a statement. A name on a PROCEDURE statement is called an entry constant; a name on a FORMAT statement is called a format constant; a name on other kinds of statements is called a label constant. A data item that has the LABEL attribute.

label constant
A name written as the label prefix of a statement (other than PROCEDURE, ENTRY, FORMAT, or PACKAGE) so that, during execution, program control can be transferred to that statement through a reference to its label prefix.

label data
A label constant or the value of a label variable.

label prefix
A label prefixed to a statement.

label variable
A variable declared with the LABEL attribute. Its value is a label constant in the program.

leading zeroes
Zeros that have no significance in an arithmetic value. All zeros to the left of the first nonzero in a number.

level number
A number that precedes a name in a DECLARE statement and specifies its relative position in the hierarchy of structure names.

level-one variable
A major structure or union name. Any unsubscripted variable not contained within a structure or union.

lexically
Relating to the left-to-right order of units.

library
An MVS partitioned data set or a CMS

MACLIB that can be used to store other data sets called members.

list-directed
The type of stream-oriented transmission in which data in the stream appears as constants separated by blanks or commas and for which formatting is provided automatically.

locator
A control block that holds the address of a variable or its descriptor.

locator/descriptor
A locator followed by a descriptor. The locator holds the address of the variable, not the address of the descriptor.

locator qualification
In a reference to a based variable, either a locator variable or function reference connected by an arrow to the left of a based variable to specify the generation of the based variable to which the reference refers. It might be an implicit reference.

locator value
A value that identifies or can be used to identify the storage address.

locator variable
A variable whose value identifies the location in main storage of a variable or a buffer. It has the POINTER or OFFSET attribute.

locked record
A record in an EXCLUSIVE DIRECT UPDATE file that has been made available to one task only and cannot be accessed by other tasks until the task using it relinquishes it.

logical level (of a structure or union member)
The depth indicated by a level number when all level numbers are in direct sequence (when the increment between successive level numbers is one).

logical operators
The bit-string operators not and exclusive-or (¬), and (&), and or (|).

loop A sequence of instructions that is executed iteratively.

lower bound
The lower limit of an array dimension.

M

main procedure
An external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. This procedure is invoked automatically as the first step in the execution of a program.

major structure
A structure whose name is declared with level number 1.

member
A structure, union, or element name in a structure or union. Data sets in a library.

minor structure
A structure that is contained within another structure or union. The name of a minor structure is declared with a level number greater than one and greater than its parent structure or union.

mode (of arithmetic data)
An attribute of arithmetic data. It is either *real* or *complex*.

multiple declaration
Two or more declarations of the same identifier internal to the same block without different qualifications. Two or more external declarations of the same identifier.

multiprocessing
The use of a computing system with two or more processing units to execute two or more programs simultaneously.

multiprogramming
The use of a computing system to execute more than one program concurrently, using a single processing unit.

multitasking
A facility that allows a program to execute more than one PL/I procedure simultaneously.

N

name Any identifier that the user gives to a variable or to a constant. An identifier appearing in a context where it is not a keyword. Sometimes called a user-defined name.

nesting
The occurrence of:

- A block within another block
- A group within another group

- An IF statement in a THEN clause or in an ELSE clause
- A function reference as an argument of a function reference
- A remote format item in the format list of a FORMAT statement
- A parameter descriptor list in another parameter descriptor list
- An attribute specification within a parenthesized name list for which one or more attributes are being factored

nonconnected storage
Storage occupied by nonconnected data items. For example, interleaved arrays and structures with inherited dimensions are in nonconnected storage.

null locator value
A special locator value that cannot identify any location in internal storage. It gives a positive indication that a locator variable does not currently identify any generation of data.

null statement
A statement that contains only the semicolon symbol (;). It indicates that no action is to be taken.

null string
A character, graphic, or bit string with a length of zero.

numeric-character data
See *decimal picture data*.

numeric picture data
Picture data that has an arithmetic value as well as a character value. This type of picture data cannot contain the characters 'A' or 'X.'

O

object A collection of data referred to by a single name.

offset variable
A locator variable with the OFFSET attribute, whose value identifies a location in storage relative to the beginning of an area.

ON-condition
An occurrence, within a PL/I program, that could cause a program interrupt. It can be the detection of an unexpected error or of an occurrence that is expected, but at an unpredictable time.

ON-statement action
The action explicitly established for a condition that is executed when the condition is raised. When the ON-statement is encountered in the flow of control for the program, it executes, establishing the action for the condition. The action executes when the condition is raised if the ON-unit is still established or a RESIGNAL statement reestablishes it. Contrast with *implicit action*.

ON-unit
The specified action to be executed when the appropriate condition is raised.

opening (of a file)
The association of a file with a data set.

operand
The value of an identifier, constant, or an expression to which an operator is applied, possibly in conjunction with another operand.

operational expression
An expression that consists of one or more operators.

operator
A symbol specifying an operation to be performed.

option A specification in a statement that can be used to influence the execution or interpretation of the statement.

P

package constant
The label prefix on a PACKAGE statement.

packed decimal
The internal representation of a fixed-point decimal data item.

padding
One or more characters, graphics, or bits concatenated to the right of a string to extend the string to a required length. One or more bytes or bits inserted in a structure or union so that the following element within the structure or union is aligned on the appropriate integral boundary.

parameter
A name in the parameter list following the PROCEDURE statement, specifying an argument that will be passed when the procedure is invoked.

parameter descriptor
The set of attributes specified for a parameter in an ENTRY attribute specification.

parameter descriptor list
The list of all parameter descriptors in an ENTRY attribute specification.

parameter list
A parenthesized list of one or more parameters, separated by commas and following either the keyword PROCEDURE in a procedure statement or the keyword ENTRY in an ENTRY statement. The list corresponds to a list of arguments passed at invocation.

partially qualified name
A qualified name that is incomplete. It includes one or more, but not all, of the names in the hierarchical sequence above the structure or union member to which the name refers, as well as the name of the member itself.

picture data
Numeric data, character data, or a mix of both types, represented in character form.

picture specification
A data item that is described using the picture characters in a declaration with the PICTURE attribute or in a P-format item.

picture specification character
Any of the characters that can be used in a picture specification.

PL/I character set
A set of characters that has been defined to represent program elements in PL/I.

PL/I prompter
Command processor program for the PLI command that checks the operands and allocates the data sets required by the compiler.

point of invocation
The point in the invoking block at which the reference to the invoked procedure appears.

pointer
A type of variable that identifies a location in storage.

pointer value
A value that identifies the pointer type.

pointer variable
A locator variable with the POINTER attribute that contains a pointer value.

precision
The number of digits or bits contained in a fixed-point data item, or the minimum number of significant digits (excluding the exponent) maintained for a floating-point data item.

prefix A label or a parenthesized list of one or more condition names included at the beginning of a statement.

prefix operator
An operator that precedes an operand and applies only to that operand. The prefix operators are plus (+), minus (-), and not (¬).

preprocessor
A program that examines the source program before the compilation takes place.

preprocessor statement
A special statement appearing in the source program that specifies the actions to be performed by the preprocessor. It is executed as it is encountered by the preprocessor.

primary entry point
The entry point identified by any of the names in the label list of the PROCEDURE statement.

priority
A value associated with a task, that specifies the precedence of the task relative to other tasks.

problem data
Coded arithmetic, bit, character, graphic, and picture data.

problem-state program
A program that operates in the problem state of the operating system. It does not contain input/output instructions or other privileged instructions.

procedure
A collection of statements, delimited by

PROCEDURE and END statements. A procedure is a program or a part of a program, delimits the scope of names, and is activated by a reference to the procedure or one of its entry names. See also *external procedure* and *internal procedure*.

procedure reference
An entry constant or variable. It can be followed by an argument list. It can appear in a CALL statement or the CALL option, or as a function reference.

program
A set of one or more external procedures or packages. One of the external procedures must have the OPTIONS(MAIN) specification in its procedure statement.

program control data
Area, locator, label, format, entry, and file data that is used to control the processing of a PL/I program.

prologue
The processes that occur automatically on block activation.

pseudovariable
Any of the built-in function names that can be used to specify a target variable. It is usually on the left-hand side of an assignment statement.

Q

qualified name
A hierarchical sequence of names of structure or union members, connected by periods, used to identify a name within a structure. Any of the names can be subscripted.

R

range (of a default specification)
A set of identifiers and/or parameter descriptors to which the attributes in a DEFAULT statement apply.

record The logical unit of transmission in a record-oriented input or output operation. A collection of one or more related data items. The items usually have different data attributes and usually are described by a structure or union declaration.

recorded key
A character string identifying a record in a direct access data set where the character string itself is also recorded as part of the data.

record-oriented data transmission
The transmission of data in the form of separate records. Contrast with *stream data transmission*.

recursive procedure
A procedure that can be called from within itself or from within another active procedure.

reentrant procedure
A procedure that can be activated by multiple tasks, threads, or processes simultaneously without causing any interference between these tasks, threads, and processes.

REFER expression
The expression preceding the keyword REFER, which is used as the bound, length, or size when the based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

REFER object
The variable in a REFER option that holds or will hold the current bound, length, or size for the member. The REFER object must be a member of the same structure or union. It must not be locator-qualified or subscripted, and it must precede the member with the REFER option.

reference
The appearance of a name, except in a context that causes explicit declaration.

relative virtual origin (RVO)
The actual origin of an array minus the virtual origin of an array.

remote format item
The letter R followed by the label (enclosed in parentheses) of a FORMAT statement. The format statement is used by edit-directed data transmission statements to control the format of data being transmitted.

repetition factor
A parenthesized unsigned integer constant that specifies:

1. The number of times the string constant that follows is to be repeated.
2. The number of times the picture character that follows is to be repeated.

repetitive specification
An element of a data list that specifies controlled iteration to transmit one or more data items, generally used in conjunction with arrays.

restricted expression
An expression that can be evaluated by the compiler during compilation, resulting in a constant. Operands of such an expression are constants, named constants, and restricted expressions.

returned value
The value returned by a function procedure.

RETURNS descriptor
A descriptor used in a RETURNS attribute, and in the RETURNS option of the PROCEDURE and ENTRY statements.

S

scalar variable
A variable that is not a structure, union, or array.

scale A system of mathematical notation whose representation of an arithmetic value is either fixed-point or floating-point.

scale factor
A specification of the number of fractional digits in a fixed-point number.

scaling factor
See *scale factor*.

scope (of a condition prefix)
The portion of a program throughout which a particular condition prefix applies.

scope (of a declaration or name)
The portion of a program throughout which a particular name is known.

secondary entry point
An entry point identified by any of the names in the label list of an entry statement.

select-group
A sequence of statements delimited by SELECT and END statements.

selection clause
A WHEN or OTHERWISE clause of a select-group.

self-defining data
An aggregate that contains data items whose bounds, lengths, and sizes are determined at program execution time and are stored in a member of the aggregate.

separator
See *delimiter*.

shift Change of data in storage to the left or to the right of original position.

shift-in
Symbol used to signal the compiler at the end of a double-byte string.

shift-out
Symbol used to signal the compiler at the beginning of a double-byte string.

sign and currency symbol characters
The picture specification characters. S, +, -, and $ (or other national currency symbols enclosed in < and >).

simple parameter
A parameter for which no storage class attribute is specified. It can represent an argument of any storage class, but only the current generation of a controlled argument.

simple statement
A statement other than IF, ON, WHEN, and OTHERWISE.

source Data item to be converted for problem data.

source key
A key referred to in a record-oriented transmission statement that identifies a particular record within a direct access data set.

source program
A program that serves as input to the source program processors and the compiler.

source variable
A variable whose value participates in some other operation, but is not modified by the operation. Contrast with *target variable*.

spill file
Data set named SYSUT1 that is used as a temporary workfile.

standard default
The alternative attribute or option assumed when none has been specified and there is no applicable DEFAULT statement.

standard file
A file assumed by PL/I in the absence of a FILE or STRING option in a GET or PUT statement. SYSIN is the standard input file and SYSPRINT is the standard output file.

standard system action
Action specified by the language to be taken for an enabled condition in the absence of an ON-unit for that condition.

statement
A PL/I statement, composed of keywords, delimiters, identifiers, operators, and constants, and terminated by a semicolon (;). Optionally, it can have a condition prefix list and a list of labels. See also *keyword statement* and *null statement*.

statement body
A statement body can be either a simple or a compound statement.

statement label
See *label constant*.

static storage allocation
The allocation of storage for static variables.

static variable
A variable that is allocated before execution of the program begins and that remains allocated for the duration of execution.

stream-oriented data transmission
The transmission of data in which the data is treated as though it were a continuous stream of individual data values in character form. Contrast with *record-oriented data transmission*.

string A contiguous sequence of characters, graphics, or bits that is treated as a single data item.

string variable
A variable declared with the BIT, CHARACTER, or GRAPHIC attribute, whose values can be either bit, character, or graphic strings.

structure
A collection of data items that need not have identical attributes. Contrast with *array*.

structure expression
An expression whose evaluation yields a structure set of values.

structure of arrays
A structure that has the dimension attribute.

structure member
See *member*.

structuring
The hierarchy of a structure, in terms of the number of members, the order in which they appear, their attributes, and their logical level.

subroutine
A procedure that has no RETURNS option in the PROCEDURE statement. Contrast with *function*.

subroutine call
An entry reference that must represent a subroutine, followed by an optional argument list that appears in a CALL statement. Contrast with *function reference*.

subscript
An element expression that specifies a position within a dimension of an array. If the subscript is an asterisk, it specifies all of the elements of the dimension.

subscript list
A parenthesized list of one or more subscripts, one for each dimension of the array, which together uniquely identify either a single element or cross section of the array.

subtask
A task that is attached by the given task or any of the tasks in a direct line from the given task to the last attached task.

synchronous
A single flow of control for serial execution of a program.

T

target Attributes to which a data item (source) is converted.

target reference
A reference that designates a receiving variable (or a portion of a receiving variable).

target variable
A variable to which a value is assigned.

task The execution of one or more procedures by a single flow of control.

task name
An identifier used to refer to a task variable.

task variable
A variable with the TASK attribute whose value gives the relative priority of a task.

termination (of a block)
Cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or to some other active block by means of a GO TO statement.

termination (of a task)
Cessation of the flow of control for a task.

truncation
The removal of one or more digits, characters, graphics, or bits from one end of an item of data when a string length or precision of a target variable has been exceeded.

type The set of data attributes and storage attributes that apply to a generation, a value, or an item of data.

U

undefined
Indicates something that a user must not do. Use of a undefined feature is likely to produce different results on different implementations of a PL/I product. In that case, the application program is in error.

union A collection of data elements that overlay each other, occupying the same storage. The members can be structures, unions, elementary variables, or arrays. They need not have identical attributes.

union of arrays
A union that has the DIMENSION attribute.

upper bound
The upper limit of an array dimension.

V

value reference
A reference used to obtain the value of an item of data.

variable
A named entity used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times.

variable reference
A reference that designates all or part of a variable.

virtual origin (VO)
The location where the element of the array whose subscripts are all zero are held. If such an element does not appear in the array, the virtual origin is where it would be held.

Z

zero-suppression characters
The picture specification characters Z and *, which are used to suppress zeros in the corresponding digit positions and replace them with blanks or asterisks respectively.

Index

Special characters

/ (forward slash) 213
*PROCESS, specifying options in 99
% statements 100
%INCLUDE statement 100, 164
  control statement 100
  source statement library 164
%NOPRINT 100
  control statement 100
%NOPRINT statement 100
%PAGE 100
  control statement 100
%PAGE statement 100
%POP statement 100
%PRINT 100
  control statement 100
%PRINT statement 100
%PROCESS, specifying options in 99
%PUSH statement 100
%SKIP 100
  control statement 100
%SKIP statement 100

A

access
  ESDS 305
  REGIONAL(1) data set 286
  relative-record data set 324
access method services
  regional data set 287
  REGIONAL(1) data set
    direct access 285
    sequential access 285
accessibility
  of Enterprise PL/I for z/OS xl
ACCT EXEC statement parameter 156
aggregate
  length table 105
AGGREGATE compiler option 7
ALIGNED compiler suboption 24
ALL option
  hooks location suboption 88
ALLOCATE statement 105
alternate ddname under z/OS UNIX, in TITLE option 213
AMP parameter 291
ANS
  compiler suboption 24
APPEND option under z/OS UNIX 215
ARCH compiler option 8, 330
argument passing
  by descriptor list 497
  by descriptor-locator 498
array descriptor
  array descriptor 500, 501
ASCII
  compiler suboption
    description 24
assembler routines
  FETCHing 183, 191
ASSERT compiler option 9
ASSIGNABLE compiler suboption 24
ATTENTION ON-units 478
attention processing
  attention interrupt,effect of 44
  ATTENTION ON-units 478
  debugging tool 478
  main description 477
attribute table 104
ATTRIBUTES option 9
automatic
  padding 171, 190
  prompting
    overriding 170, 188
    using 169, 188
  restart
    after system failure 481
    checkpoint/restart facility 479
    within a program 481
auxiliary storage for sort 352
avoiding calls to library routines 341

B

BACKREG compiler option 10
batch compile
  OS/390 159, 161
BIFPREC compiler option 10
BIN1ARG compiler suboption 25
BKWD option 228, 299
BLANK compiler option 11
BLKOFF compiler option 12
BLKSIZE
  ENVIRONMENT 228
    comparison with DCB subparameter 230
    for record I/O 232
  option of ENVIRONMENT
    for stream I/O 246
  subparameter 225
block
  and record 221
  size
    maximum 232
    object module 163
    PRINT files 255
    record length 233
    regional data sets 288
    specifying 221
BRACKETS 12
BRACKETS compiler option 12
BUFFERS option
  for stream I/O 246
BUFND option 299
BUFNI option 299
BUFSIZE option under z/OS UNIX 216
BUFSP option 300
BYADDR
  description 332
BYADDR *(continued)*
  effect on performance 333
  using with DEFAULT option 25
BYVALUE
  description 332
  effect on performance 333
  using with DEFAULT option 25

C

C routines
  FETCHing 183
capacity record
  REGIONAL(1) 283
carriage control character 51, 254
carriage return-line feed (CR - LF) 220
CASERULES compiler option 13
cataloged procedure
  compile and bind 151
  compile only 150
  compile, bind, and run 153
  compile, input data for 153
  description of 149
  invoking 155
  listing 155
  modifying
    DD statement 157
    EXEC statement 156
  multiple invocations 155
  under OS/390
    IBM-supplied 149
    to invoke 155
    to modify 156
CEESTART compiler option 13
character string attribute table 104
characters
  carriage control 51, 254
  print control 51, 254
CHECK compiler option 14
checkpoint data
  for sort 356
checkpoint data, defining, PLICKPT built-in suboption 480
checkpoint/restart
  deferred restart 482
  PLICANC statement 482
checkpoint/restart facility
  CALL PLIREST statement 481
  checkpoint data set 480
  description of 479
  modify activity 482
  PLICKPT built-in subroutine 479
  request checkpoint record 479
  request restart 481
  RESTART parameter 482
  return codes 479
CHKPT sort option 350
CICS
  preprocessor options 146
  support 145
CKPT sort option 350

© Copyright IBM Corp. 1999, 2017

CMPAT compiler option 15
COBOL
  map structure 105
CODE subparameter 225
CODEPAGE compiler option 16
coding
  CICS statements 146
  improving performance 336
  SQL statements 129
comments
  within options 98
communications area, SQL 129
compilation
  user exit
    activating 486
    customizing 486
    example 490
    IBMUEXIT 485
    procedures 483
compile and bind, input data for 151
COMPILE compiler option 17
compile-time options
  under z/OS UNIX 160
compiler
  % statements 100
  DBCS identifier 38
  descriptions of options 3
  general description of 159
  graphic string constant 38
  invoking 159
  JCL statements, using 161
  listing
    aggregate length table 105
    attribute table 104
    block level 104
    cross-reference table 105
    DO-level 104
    file reference table 109
    heading information 102
    include source program 43
    input to compiler 103
    input to preprocessor 103
    messages 110
    printing options 164
    return codes 111
    SOURCE option program 103
    source program 84
    stack storage used 85
    statement offset addresses 106
    SYSPRINT 164
    using 102
  mixed string constant 38
  PROCESS statement 99
  reduce storage requirement 59
  severity of error condition 17
  temporary workfile (SYSUT1) 164
  under OS/390 batch 161
compiler options
  abbreviations 7
  AGGREGATE 7
  ARCH 8, 330
  ASSERT 9
  ATTRIBUTES 9
  BACKREG 10
  BIFPREC 10
  BLANK 11
  BLKOFF 12
compiler options *(continued)*
  CASERULES 13
  CEESTART 13
  CHECK 14
  CMPAT 15
  CODEPAGE 16
  COMPILE 17
  COPYRIGHT 17
  CSECT 18
  CSECTCUT 18
  CURRENCY 19
  DBCS 19
  DD 19
  DDSQL 20
  default 3
  DEFAULT 22, 332
  DEPRECATE 31
  DEPRECATENEXT 32
  description of 3
  DISPLAY 33
  DLLINIT 33
  EXIT 33
  EXTRN 34
  FILEREF 34
  FLAG 35
  FLOAT 35
  FLOATINMATH 36
  GOFF 37
  GONUMBER 38, 329
  GRAPHIC 38
  IGNORE 39
  INCAFTER 40
  INCDIR 40
  INCLUDE 40
  INSOURCE 43
  INTERRUPT 44
  LANGLVL 45
  LIMITS 45
  LINECOUNT 47
  LINEDIR 47
  LIST 47
  LISTVIEW 47
  MACRO 49
  MAP 50
  MARGINI 50
  MARGINS 50
  MAXBRANCH 51
  MAXGEN 52
  MAXMEM 52
  MAXMSG 53
  MAXNEST 53
  MAXSTMT 54
  MAXTEMP 54
  MDECK 54
  MSGSUMMARY 55
  NAME 55
  NAMES 56
  NATLANG 56
  NEST 56
  NOMARGINS 50
  NOT 57
  NUMBER 57
  OBJECT 58
  OFFSET 58
  OFFSETSIZE 58
  ONSNAP 59
  OPTIMIZE 59, 329
compiler options *(continued)*
  OPTIONS 60
  OR 61
  PP 61
  PPCICS 62
  PPINCLUDE 63
  PPLIST 63
  PPMACRO 64
  PPSQL 64
  PPTRACE 65
  PRECTYPE 65
  PREFIX 65, 331
  PROCEED 66
  QUOTE 67
  REDUCE 67, 330
  RENT 68
  RESEXP 69
  RESPECT 70
  RTCHECK 70
  RULES 70, 330
  SEMANTIC 83
  SERVICE 83
  SOURCE 84
  SPILL 84
  STATIC 84
  STDSYS 85
  STMT 85
  STORAGE 85
  STRINGOFGRAPHIC 86
  SYNTAX 86
  SYSPARM 87
  SYSTEM 87
  TERMINAL 88
  TEST 88
  UNROLL 91
  USAGE 92
  WIDECHAR 93
  WINDOW 93
  WRITABLE 93
  XINFO 95
  XML 45, 97
  XREF 97
compiling
  under z/OS UNIX 159
concatenating
  data sets 210
  external references 207
COND EXEC statement parameter 156
conditional compilation 17
conditional subparameter 225
configuration file
  include preprocessor 118
  macro preprocessor 122
CONNECTED compiler suboption
  description 26
  effect on performance 334
CONSECUTIVE
  option of ENVIRONMENT 247, 269
consecutive data sets
  controlling input from the terminal
    capital and lowercase letters 262
    condition format 259
    COPY option of GET
      statement 265
    defining QSAM files 262
    end-of-file 262
    format of data 261

consecutive data sets *(continued)*
controlling input from the terminal *(continued)*
stream and record files 261
controlling output to the terminal
conditions 265
format of PRINT files 265
output from the PUT EDIT command 267
stream and record files 265
defining and using 245
input from the terminal 259
output to the terminal 265
record-oriented data transmission
accessing and updating a data set 273
creating a data set 272
defining files 268
specifying ENVIRONMENT options 268
statements and options allowed 267
record-oriented I/O 267
stream-oriented data transmission 245
accessing a data set 253
creating a data set 249
defining files 245, 246
specifying ENVIRONMENT options 246
using PRINT files 254
using SYSIN and SYSPRINT files 259
control
area 292
CONTROL option
EXEC statement 165
interval 292
control blocks
function-specific 484
global control 487
control characters
carriage 51, 254
print 51, 254
COPY option 265
COPYRIGHT compiler option 17
COPYRIGHT option 475
counter records, SYSADATA 507
cross-reference table
compiler listing 105
using XREF option 104
CSECT compiler option 18
CSECTCUT compiler option 18
CTLASA and CTL360 options
ENVIRONMENT option
for consecutive data sets 270
SCALARVARYING 235
CURRENCY compiler option 19
customizing
user exit
modifying SYSUEXIT 486
structure of global control blocks 484
writing your own compiler exit 487
CYLOFL subparameter
DCB parameter 225

D

data
conversion under z/OS UNIX 212
files
creating under z/OS UNIX 214
sort program 356
PLISRT(x) command 361
sorting
description of 345
types
equivalent Java and PL/I 400
equivalent SQL and PL/I 136
data definition (DD) information under z/OS UNIX 212
data set
associating PL/I files with
closing a file 227
opening a file 226
specifying characteristics in the ENVIRONMENT attribute 228
associating several data sets with one file 209
blocks and records 221
checkpoint 480
conditional subparameter characteristics 225
consecutive stream-oriented data 245
data set control block (DSCB) 224
ddnames 162
defining for dump
DD statement 468
logical record length 468
defining relative-record 321
direct 223, 224
dissociating from a file 227
system-determined block size 227
dissociating from PL/I file 210
establishing characteristics 221
indexed
sequential 223
information interchange codes 221
label modification 226
labels 224, 239
libraries
extracting information 244
SPACE parameter 240
types of 239
use 240
organization
conditional subparameters 225
data definition (DD) statement 224
types of 223
partitioned 239
record format defaults 230
record formats
fixed-length 222
undefined-length 223
variable-length 222
records 221
regional 279
REGIONAL(1) 282
accessing and updating 285
creating 283
sequential 223
sort program
checkpoint data set 356
data set *(continued)*
sort program *(continued)*
input data set 356
output data set 356
sort work data set 355
sorting 355
SORTWK 352
source statement library 164
SPACE parameter 162
stream files 245
temporary 164
to establish characteristics 221
types of
comparison 236
used by PL/I record I/O 236
unlabeled 224
using 205
VSAM
blocking 292
data set type 295
defining 302
defining files 298
dummy data set 296
indexed data set 306
keys 294
mass sequential insert 312
organization 292
running a program 291
specifying ENVIRONMENT options 298
VSAM option 301
VSAM.
performance options 301
data set under OS/390
associating one data set with several files 209
concatenating 210
HFS 210
data set under z/OS UNIX
associating a PL/I file with a data set
how PL/I finds data sets 214
using environment variables under 212
using the TITLE option of the OPEN statement 213
using unassociated files 214
DD_DDNAME environment variable 211
default identification 211
establishing a path 214
establishing characteristics
DD_DDNAME environment variable 215
extending on output 215
maximum number of regions 218
number of regions 218
recreating output 215
data sets
allocating files 205
associating data sets with files 207
closing 227
defining data sets under OS/390 207
data-directed I/O 336
coding for performance 336
DBCS compiler option 19
DBCS identifier compilation 38
DCB subparameter 230

DCB subparameter *(continued)*
equivalent ENVIRONMENT options 230
main discussion of 225
overriding in cataloged procedure 157
regional data set 288
DD (data definition) information under z/OS UNIX 212
DD compiler option 19
DD information under z/OS UNIX
TITLE statement 213
DD statement 224
%INCLUDE 101
add to cataloged procedure 157
cataloged procedure, modifying 157
checkpoint/restart 479
create a library 240
modifying cataloged procedure 156
OS/390 batch compile 162
regional data set 288
standard data set 162
input (SYSIN) 163
output (SYSLIN, SYSPUNCH) 163
DD Statement
modify cataloged procedure 157
DD_DDNAME environment variables
alternate ddname under z/OS UNIX 213
APPEND 215
DELAY 216
DELIMIT 217
LRECL 217
LRMSKIP 217
PROMPT 217
PUTPAGE 217
RECCOUNT 218
RECSIZE 218
SAMELINE 218
SKIP0 219
specifying characteristics under z/OS UNIX 215
TYPE 219
ddname
%INCLUDE 101
standard data sets 162
DDSQL compiler option 20
deblocking of records 221
declaration
of files under OS/390 205, 207
declaring
host variables, SQL preprocessor 133
DECOMP 22
DECOMP compiler option 22
DEFAULT compiler option
description and syntax 22
suboptions
ALIGNED 24
ASCII or EBCDIC 24
ASSIGNABLE or NONASSIGNABLE 24
BIN1ARG or NOBIN1ARG 25
BYADDR or BYVALUE 25
CONNECTED or NONCONNECTED 26
DESCLIST or DESCLOCATOR 26
DEFAULT compiler option *(continued)*
suboptions *(continued)*
DESCRIPTOR or NODESCRIPTOR 26
DUMMY 26
E 27
EVENDEC or NOEVENDEC 27
HEXADEC 27
IBM or ANS 24
INITFILL or NOINITFILL 27
INLINE or NOINLINE 28
LINKAGE 28
LOWERINC | UPPERINC 28
NATIVE or NONNATIVE 28
NATIVEADDR or NONNATIVEADDR 29
NULLSTRADDR or NONULLSTRADDR 29
NULLSTRPTR 29
NULLSYS or NULL370 29
ORDER or REORDER 30
ORDINAL(MIN | MAX) 30
OVERLAP | NOOVERLAP 30
PSEUDODUMMY or NOPSEUDODUMMY 30
RECURSIVE or NONRECURSIVE 30
RETCODE 30
RETURNS 31
SHORT 31
deferred restart 482
define data set
associating several data sets with one file 209
associating several files with one data set 209
closing a file 227
concatenating several data sets 210
ENVIRONMENT attribute 228
ESDS 304
opening a file 226
system-determined block size 227
specifying characteristics 228
define file
associating several files with one data set 209
closing a file 227
concatenating several data sets 210
ENVIRONMENT attribute 228
opening a file 226
system-determined block size 227
regional data set 281
ENV options 281
keys 282
specifying characteristics 228
VSAM data set 298
define file under OS/390
associating several data sets with one file 209
DEFINED
versus UNION 339
DELAY option under z/OS UNIX
description 216
DELIMIT option under z/OS UNIX
description 217
DEPRECATE compiler option 31
DEPRECATENEXT compiler option 32
DESCLIST compiler suboption 26
DESCLOCATOR compiler suboption 26
descriptor 497
descriptor area, SQL 130
DESCRIPTOR compiler option
effect on performance 334
DESCRIPTOR compiler suboption
description 26
descriptor list, argument passing 497
descriptor-locator, argument passing 498
DFSORT 345
direct data sets 223, 224
DIRECT file
indexed ESDS with VSAM
accessing data set 310
updating data set 312
RRDS
access data set 324
DISP parameter
consecutive data sets 274
to delete a data set 239
DISPLAY compiler option 33
DLL
DYNAM=DLL linker option 175
linking considerations and side-decks 167, 187
RENT compiler option and fetching 174
DLLINIT compiler option 33
DSA
saved in PLIDUMP built-in subroutine for each block 469
DSCB (data set control block) 224, 241
DSNAME parameter
for consecutive data sets 274
DSORG subparameter 225
DUMMY compiler suboption 26
dummy records
REGIONAL(1) data set 283
VSAM 296
dump
calling PLIDUMP 467
defining data set for
DD statement 468
logical record length 468
identifying beginning of 468
PLIDUMP built-in subroutine 467
producing z/OS Language Environment dump 467
SNAP 468
DYNALLOC sort option 350

E

E compiler message 110
E compiler suboption 27
E15 input handling routine 357
E35 output handling routine 360
EBCDIC
compiler suboption 24
EBCDIC (Extended Binary Coded Decimal Interchange Code) 221
embedded
CICS statements 146
SQL statements 131
ENDFILE
under OS/390 171, 190

Enterprise PL/I for z/OS
accessibility xl
Enterprise PL/I library
Enterprise PL/I for z/OS library xv
Language Environment library xvi
entry-sequenced data set
defining 304
VSAM 293
loading an ESDS 303
SEQUENTIAL file 303
statements and options 302
ENVIRONMENT attribute
list 228
specifying characteristics under z/OS UNIX
BUFSIZE 216
ENVIRONMENT options
BLKSIZE option
comparison with DCB subparameter 230
CONSECUTIVE 247, 269
CTLASA and CTL360 270
comparison with DCB subparameter 230
equivalent DCB subparameters 230
GRAPHIC option 248
KEYLENGTH option
comparison with DCB subparameter 230
LEAVE and REREAD 271
organization options 230
record format options 247
RECSIZE option
comparison with DCB subparameter 230
record format 248
usage 248
regional data set 281
VSAM
BKWD option 299
BUFND option 299
BUFNI option 299
BUFSP option 300
GENKEY option 300
PASSWORD option 300
REUSE option 300
SKIP option 301
VSAM option 301
environment variables
include preprocessor 118
macro preprocessor 122
setting under z/OS UNIX 237
EQUALS sort option 350
error
severity of error compilation 17
error devices
redirecting 238
ESDS (entry-sequenced data set)
defining 304
nonunique key alternate index path 314
unique key alternate index path 313
VSAM 293
loading 303
statements and options 302
EVENDEC compiler suboption 27
examples
calling PLIDUMP 467
EXEC SQL statements 123
EXEC statement
cataloged procedure, modifying 156
compiler 162
introduction 162
maximum length of option list 165
minimum region size 162
modify cataloged procedure 156
OS/390 batch compile 159, 162
PARM parameter 165
to specify options 165
Exit (E15) input handling routine 357
Exit (E35) output handling routine 360
EXIT compiler option 33
export command 215
EXPORTALL 34
EXPORTALL compiler option 34
extended binary coded decimal interchange code (EBCDIC) 221
EXTERNAL attribute 104
external references
concatenating names 207
EXTRN compiler option 34

F

F option of ENVIRONMENT
for record I/O 230
for stream I/O 246, 247
F-format records 222
FB option of ENVIRONMENT
for record I/O 230
for stream I/O 246, 247
FB-format records 222
FBS option of ENVIRONMENT
for record I/O 230
for stream I/O 246
FETCH
assembler routines 183, 191
Enterprise PL/I routines 174, 190
OS/390 C routines 183
PL/I MAIN Routines 182, 191
field for sorting 349
file
allocating files 205
associating data sets with files 207
closing 227
defining data sets under OS/390 207
establishing characteristics 221
FILE attribute 104
file records, SYSADATA 508
FILEREF compiler option 34
filespec 215
FILLERS, for tab control table 257
FILSZ sort option 350
filtering messages 485
FIXED
TYPE option under z/OS UNIX 220
fixed-length records 222
FLAG compiler option 35
flags, specifying compile-time options 161
FLOAT option 35
FLOATINMATH compiler option 36
flowchart for sort 357
format notation, rules for xvii
forward slash (/) 213
FS option of ENVIRONMENT
for record I/O 230
for stream I/O 246
FUNC subparameter
usage 225
Functions
using C functions with ILC 369

G

GENKEY option
key classification 234
usage 228
VSAM 298
GET DATA statement 171, 189
GET EDIT statement 171, 190
GET LIST statement 171, 189
global control blocks
writing the initialization procedure 487
writing the message filtering procedure 487
writing the termination procedure 489
GOFF compiler option 37
GONUMBER compiler option 329
definition 38
GOTO statements 337
graphic data 245
GRAPHIC option
compiler 38
of ENVIRONMENT 228, 248
stream I/O 246
graphic string constant compilation 38

H

handling routines
data for sort
input (sort exit E15) 357
output (sort exit E35) 360
PLISRTB 362
PLISRTC 363
PLISRTD 364
to determine success 355
variable length records 365
HEADER 39
HEADER compiler option 39
header label 224
HEXADEC compiler suboption 27
hook
location suboptions 88
host
structures 139
variables, using in SQL statements 133

I

I compiler message 110
IBM compiler suboption 24
IBMUEXIT compiler exit 485
IBMZC cataloged procedure 150
IBMZCB cataloged procedure 151

IBMZCBG cataloged procedure 153
identifiers
not referenced 9
source program 9
IEC225I 225, 288
IGNORE option 39
ILC
linkage considerations 375
with C 367
data types 367
Enum data type 369
File type 369
Functions returning ENTRYs 374
parameter matching 370
Redirecting C standard streams 378
sharing input 378
sharing output 377
string parameter type matching 373
structure data type 368
Using the ATTACH statement 378
improving application performance 329
INCAFTER compiler option 40
INCDIR compiler option 40
INCLUDE compiler option 40
include preprocessor
environment variables 118
syntax 117
using configuration file 118
INCLUDE statement
compiler 101
INDEXAREA option 228
indexed data sets
indexed sequential data set 223
indexed ESDS (entry-sequenced data set)
DIRECT file 310
loading 308
SEQUENTIAL file 310
indicator variables, SQL 139
information interchange codes 221
INITFILL compiler suboption 27
initial volume label 224
initialization procedure of compiler user exit 487
INLINE compiler suboption 28
input
data for PLISRTA 361
data for sort 356
defining data sets for stream files 245, 246
redirecting 238
routines for sort program 356
SEQUENTIAL 272
skeletal code for sort 360
sort data set 356
input/output
compiler
data sets 163
data for compile and bind 151
in cataloged procedures 150
OS/390, punctuating long lines 170, 189
skeletal code for sort 358
sort data set 356
INSOURCE option 43
Inter Language Communication
linkage considerations 375
Redirecting C standard streams 378
Using the ATTACH statement 378
with C 367
data types 367
Enum data type 369
File type 369
Functions returning ENTRYs 374
parameter matching 370
sharing input 378
sharing output 377
string parameter type matching 373
structure data type 368
using C functions 369
interactive program
attention interrupt 44
interblock gap (IBG) 221
interchange codes 221
INTERNAL attribute 104
INTERRUPT compiler option 44
interrupts
attention interrupts under interactive system 44
ATTENTION ON-units 478
debugging tool 478
main description 477
invoking
cataloged procedure 155
link-editing multitasking programs 156
multiple invocations 155

J

Java 382, 383, 385, 386, 387, 388, 390, 393, 394, 395, 396, 399
JAVA 381
Java code, compiling 383, 387, 393, 396
Java code, writing 382, 386, 390, 395
JCL (job control language)
improving efficiency 149
using during compilation 161
jni
JNI sample program 382, 386, 390, 395

K

key indexed VSAM data set 295
key-sequenced data sets
accessing with a DIRECT file 310
accessing with a SEQUENTIAL file 310
loading 308
statements and options for 306
KEYLEN subparameter 225
KEYLENGTH option 228, 236
KEYLOC option
usage 228
keys
alternate index
nonunique 313
unique 313
REGIONAL(1) data set 282
keys *(continued)*
dummy records 283
VSAM
indexed data set 295
relative byte address 295
relative record number 295
KEYTO option
under VSAM 303
KSDS (key-sequenced data set)
define and load 308
unique key alternate index path 315
updating 310
VSAM
DIRECT file 310
loading 308
SEQUENTIAL file 310
KSDS with VSAM
indexed ESDS with VSAM
access data set 310
accessing data set 310
updating data set 312

L

label
for data sets 224
LANGLVL compiler option 45
Language Environment library xvi
LEAVE and REREAD options
ENVIRONMENT option
for consecutive data sets 271
length of record
specifying under z/OS UNIX 218
library
compiled object modules 242
creating a data set library 240
creating and updating a library member 241
directory 240
extracting information from a library directory 244
general description of 224
how to use 240
information required to create 240
placing a load module 242
source statement 164
source statement library 159
SPACE parameter 240
structure 244
system procedure (SYS1.PROCLIB) 239
types of 239
using 239
LIMCT subparameter 225, 288
LIMITS compiler option 45
line
length 255
numbers in messages 38
line feed (LF)
definition 220
LINE option 247, 255
LINECOUNT compiler option 47
LINEDIR compiler option 47
LINESIZE option
for tab control table 257
OPEN statement 248

link-editing
description of 167, 187
LINKAGE compiler suboption
effect on performance 334
syntax 28
linkage considerations with ILC 375
LIST compiler option 47
listing
cataloged procedure 155
compiler
aggregate length table 105
ATTRIBUTE and cross-reference table 104
ddname list 3
file reference table 109
heading information 102
messages 110
options 103
preprocessor input 103
return codes 111
SOURCE option program 103
statement nesting level 104
statement offset addresses 106
storage offset listing 108
OS/390 batch compile 159, 164
source program 84
statement offset address 106
storage offset listing 108
SYSPRINT 164
LISTVIEW compiler option 47
literal records, SYSADATA 508
logical not 57
logical or 61
loops
control variables 337
LOWERINC compiler suboption 28
LP 49
LP compiler option 49
LRECL option under z/OS UNIX 217
LRECL subparameter 221, 225
LRMSKIP option under z/OS UNIX 217

M

MACRO option 49
macro preprocessor
environment variables 122
macro definition 118
using configuration file 122
main storage for sort 351
MAP compiler option 50
MARGINI compiler option 50
MARGINS compiler option 50
mass sequential insert 312
MAXBRANCH compiler option 51
MAXGEN compiler option 52
MAXMEM compiler option 52
MAXMSG compiler option 53
MAXSTMT compiler option 54
MAXTEMP compiler option 54
MDECK compiler option
description 54
message
compiler list 110
printed format 259
runtime message line numbers 38
message records, SYSADATA 508
messages
filter function 487
modifying in compiler user exit 486
mixed string constant compilation 38
MODE subparameter
usage 225
module
create and store object module 58
MSGSUMMARY compiler option 55
multiple
invocations
cataloged procedure 155

N

NAME compiler option 55
named constants
defining 340
versus static variables 340
NAMES compiler option 56
NATIVE compiler suboption
description 28
NATIVEADDR compiler suboption 29
NATLANG compiler option 56
negative value
block-size 232
record length 231
NEST option 56
NOBIN1ARG compiler suboption 25
NODESCRIPTOR compiler suboption 26
NOEQUALS sort option 350
NOEVENDEC compiler suboption 27
NOINITFILL compiler suboption 27
NOINLINE compiler suboption 28
NOINTERRUPT compiler option 44
NOMAP option 105
NOMARGINS compiler option 50
NONASSIGNABLE compiler suboption 24
NONCONNECTED compiler suboption 26
NONE, hooks location suboption 88
NONNATIVE compiler suboption 28
NONNATIVEADDR compiler suboption 29
NONRECURSIVE compiler suboption 30
NONULLSTRADDR compiler suboption 29
NOOVERLAP compiler suboption 30
effect on performance 335
NOPSEUDODUMMY compiler suboption 30
NOSYNTAX compiler option 86
NOT compiler option 57
note statement 110
NTM subparameter
usage 225
NULL370 compiler suboption 29
NULLDATE 57
NULLDATE compiler option 57
NULLSTRADDR compiler suboption 29
NULLSTRPTR compiler suboption 29
NULLSYS compiler suboption 29
NUMBER compiler option 57

O

object
module
create and store 58
record size 163
OBJECT compiler option
definition 58
offset
of tab count 257
table 106
OFFSET compiler option 58
OFFSETSIZE compiler option 58
ONSNAP compiler option 59
OPEN statement
subroutines of PL/I library 226
TITLE option 226
Operating system
data definition (DD) information under z/OS UNIX 212
OPTCD subparameter 225
optimal coding
coding style 336
compiler options 329
OPTIMIZE compiler option 329
OPTIMIZE option 59
options
for compiling 103
for creating regional data set 279
saved options string in PLIDUMP 476
specifying comments within 98
specifying strings within 98
to specify for compilation 165
OPTIONS option 60
options record, SYSADATA 507
options under z/OS UNIX
DD_DDNAME environment variables
APPEND 215
DELAY 216
DELIMIT 217
LRECL 217
LRMSKIP 217
PROMPT 217
PUTPAGE 217
RECCOUNT 218
RECSIZE 218
SAMELINE 218
SKIP0 219
TYPE 219
PL/I ENVIRONMENT attribute
BUFSIZE 216
using
DD information 213
TITLE 213
OR compiler option 61
ORDER compiler suboption
description 30
effect on performance 335
ORDINAL compiler suboption 30
ordinal element records, SYSADATA 510
ordinal type records, SYSADATA 509
ORGANIZATION option 236
usage 228
OS/390
batch compilation
DD statement 162

OS/390 *(continued)*
batch compilation *(continued)*
EXEC statement 162, 165
listing (SYSPRINT) 164
source statement library (SYSLIB) 164
specifying options 165
temporary workfile (SYSUT1) 164
general compilation 159
long input lines 170, 189
output
data for PLISRTA 361
data for sort 356
defining data sets for stream files 245, 246
limit preprocessor output 54
redirecting 238
routines for sort program 356
SEQUENTIAL 272
skeletal code for sort 360
sort data set 356
SYSLIN 163
SYSPUNCH 163
OVERLAP compiler suboption 30

P

PACKAGEs versus nested PROCEDUREs 338
PAGE option 247
PAGELENGTH, for tab control table 257
PAGESIZE, for tab control table 257
parameter passing
argument passing 497
CMPAT(LE) descriptors 500
CMPAT(V*) descriptors 498
PARM parameter
for cataloged procedure 156
specify options 165
passing an argument 497
PASSWORD option 300
performance
VSAM options 301
performance improvement
coding for performance
avoiding calls to library routines 341
DATA-directed input and output 336
DEFINED versus UNION 339
GOTO statements 337
input-only parameters 336
loop control variables 337
named constants versus static variables 340
PACKAGEs versus nested PROCEDUREs 338
preloading calls to library routines 343
REDUCIBLE functions 339
string assignments 337
selecting compiler options
ARCH 330
DEFAULT 332
GONUMBER 329
OPTIMIZE 329
PREFIX 331
performance improvement *(continued)*
selecting compiler options *(continued)*
REDUCE 330
RULES 330
PL/I
compiler
user exit procedures 484
files
associating with a data set under z/OS UNIX 211
PL/I code, compiling 385, 390, 394, 399
PL/I code, linking 385, 390, 394, 399
PL/I code, writing 383, 388, 393, 396
PL/I dynamic allocation
access HFS files under z/OS 210
allocate data sets 205
associating data sets with files under z/OS UNIX 211
define files
QSAM files 262
REGIONAL(1) data sets 281
stream files 246
VSAM file 291
PL/I MAIN Routines
FETCHing 182, 191
PLICANC statement, and checkpoint/request 482
PLICKPT built-in subroutine 479
PLIDUMP built-in subroutine
calling to produce a z/OS Language Environment dump 467
DSA saved for each block 469
H option 468
output
locating variables in 469
program unit name in dump traceback table 469
saved load module timestamp 475
saved options string 476
syntax of 467
user-identifier 468
variables
locating AUTOMATIC variables in 469
locating CONTROLLED variables in 471
locating in PLIDUMP output 469
locating STATIC variables in 470
PLIREST statement 481
PLIRETC built-in subroutine
return codes for sort 355
PLISAXA 403, 404
PLISAXB 403, 404
PLISAXC 433, 434
PLISAXD 433, 434
PLISRTA interface 361
PLISRTB interface 362
PLISRTC interface 363
PLISRTD interface 364
PLITABS external structure
control section 258
declaration 168
PLIXOPT variable 167, 187
PP compiler option 61
PPCICS compiler option 62
PPINCLUDE compiler option 63
PPLIST compiler option 63
PPMACRO compiler option 64
PPSQL compiler option 64
PPTRACE compiler option 65
PRECTYPE compiler option 65
PREFIX compiler option 65, 331
using default suboptions 331
preloading library routines 343
preprocessing
%INCLUDE statement 101
input 103
limit output to 80 bytes 54
source program 49
with MACRO 49
preprocessor options
CCSID0 126
CODEPAGE 127
DEPRECATE 127
EMPTYDBRM 127
HOSTCOPY 128
INCONLY 128
WARNDECP 128
preprocessors
available with PL/I 117
CICS options 146
include 117
macro preprocessor 118
SQL options 125
SQL preprocessor 123
print
PRINT file
format 265
line length 255
stream I/O 254
print control character 51, 254
PRINT file
formatting conventions 168, 188
punctuating output 168
record I/O 274
procedure
cataloged, using under OS/390 149
compile and bind (IBMZCB) 151
compile only (IBMZC) 150
compile, bind, and run (IBMZCBG) 153
PROCEED compiler option 66
PROCESS statement
description 99
override option defaults 165
program unit name in dump traceback table 469
PROMPT option under z/OS UNIX 217
prompting
automatic, overriding 170, 188
automatic, using 169, 188
PRTSP subparameter
usage 225
PSEUDODUMMY compiler suboption 30
punctuation
automatic prompting
overriding 170, 188
using 169, 188
OS/390
automatic padding for GET EDIT 171, 190
continuation character 170, 189

punctuation *(continued)*
OS/390 *(continued)*
entering ENDFILE at terminal 171, 190
GET DATA statement 171, 189
GET LIST statement 171, 189
SKIP option 171, 190
output from PRINT files 168
PUT EDIT command 267
PUTPAGE option under z/OS UNIX 217

Q

QUOTE compiler option 67

R

REAL attribute 104
RECCOUNT option under z/OS UNIX 218
RECFM subparameter 225
in organiza tion of data set 225
usage 225
record
checkpoint 479
data set 480
deblocking 221
maximum size for compiler input 163
sort program 350
record format
fixed-length records 222
options 247
stream I/O 253
to specify 268
types 221
undefined-length records 223
variable-length records 222
record I/O
data set
access 273
consecutive data sets 274
create 272
types of 236
data transmission 267
ENVIRONMENT option 268
format 230
record format 268
record length
regional data sets 279
specify 221
value of 231
RECORD statement 350
recorded key
regional data set 282
records
length under z/OS UNIX 218
RECSIZE option
consecutive data set 248
defaults 248
definition 231
description under z/OS UNIX 218
for stream I/O 246, 248
RECURSIVE compiler suboption 30
REDUCE compiler option 67
REDUCE compiler option *(continued)*
effect on performance 330
reduce storage requirement 59
REDUCIBLE functions 339
region
REGION parameter 156
size, EXEC statement 162
regional data sets
DD statement
accessing 289
creating 288
defining files for
regional data set 281
specifying ENVIRONMENT options 281
using keys 282
operating system requirement 287
REGIONAL(1) data set
accessing and updating 285
creating 283
using 282
REGIONAL option of ENVIRONMENT 282
REGIONAL(1) data sets
defining files for
regional data set 281
regions under z/OS UNIX 218
relative byte address (RBA) 295
relative record number 295
relative-record data sets
accessing with a DIRECT file 324
accessing with a SEQUENTIAL file 324
loading 321
statements and options for 320
RENT compiler option 68
REORDER compiler suboption
description 30
effect on performance 335
RESEXP compiler option 69
RESPECT compiler option 70
restarting
requesting 481
RESTART parameter 482
to request
automatic after system failure 481
automatic within a program 481
deferred restart 482
to cancel 481
to modify 482
RETCODE compiler suboption 30
return code
checkpoint/restart routine 479
PLIRETC 355
return codes
in compiler listing 111
RETURNS compiler suboption 31, 335
REUSE option 228, 300
RRDS (relative record data set)
define 322
load with VSAM 321
updating 324
VSAM
DIRECT file 324
loading 321
SEQUENTIAL file 324
RTCHECK compiler option 70
RULES compiler option 70
effect on performance 330
run time
message line numbers 38
run-time
OS/390 considerations
automatic prompting 169
formatting conventions 168, 188
GET EDIT statement 171, 190
GET LIST and GET DATA statements 171, 189
punctuating long lines 170, 189
SKIP option 171
using PLIXOPT 167, 187
runtime
OS/390 considerations
automatic prompting 188
SKIP option 190

S

S compiler message 110
SAMELINE option under z/OS UNIX 218
sample program, running 385, 390, 395, 399
saved options string in PLIDUMP 476
SAX parser 403, 433
SCALARVARYING option 235
SEMANTIC compiler option 83
sequential access
REGIONAL(1) data set 285
sequential data set 223
SEQUENTIAL file
ESDS with VSAM
defining and loading 304
updating 305
indexed ESDS with VSAM
access data set 310
RRDS, access data set 324
serial number volume label 224
SERVICE compiler option 83
shift code compilation 38
SHORT compiler suboption 31
SKIP option 301
in stream I/O 247
under OS/390 171, 190
SKIP0 option under z/OS UNIX 219
SKIPREC sort option 350
sorting
assessing results 355
calling sort 352
CHKPT option 350
choosing type of sort 346
CKPT option 350
data 345
data input and output 356
description of 345
DYNALLOC option 350
E15 input handling routine 357
EQUALS option 350
FILSZ option 350
maximum record length 351
PLISRT 345
PLISRTA(x) command 361, 365
preparation 345
RECORD statement 357

sorting *(continued)*
RETURN statement 357
SKIPREC option 350
SORTCKPT 356
SORTCNTL 356
SORTIN 356
sorting field 349
SORTLIB 355
SORTOUT 356
SORTWK 352, 355
storage
auxiliary 352
main 351
writing input/output routines 356
source
key
in REGIONAL(1) data sets 282
listing
location 50
program
compiler list 103
data set 163
identifiers 9
included in compiler list 43
list 84
preprocessor 49
shifting outside text 50
SOURCE compiler option 84
source records, SYSADATA 515
source statement library 164
SPACE parameter
library 240
standard data sets 162
specifying compile-time options
using flags under 161
SPILL compiler option 84
spill file 164
SQL preprocessor
communications area 129
descriptor area 130
EXEC SQL statements 123
options 125
using host structures 139
using host variables 132
using IBMUEXIT with 144
using indicator variables 139
SQL preprocessor options 126, 127, 128
SQLCA 129
SQLDA 130
STACK subparameter
usage 225
standard data set 162
standard files (SYSPRINT and SYSIN) 237
statement
nesting level 104
offset addresses 106
statements 100
STATIC compiler option 84
STDSYS compiler option 85
step abend 225
STMT compiler option 85
STMT suboption of test 88
storage
blocking print files 255
library data sets 240
report in listing 85
storage *(continued)*
sort program 351
auxiliary storage 352
main storage 351
standard data sets 162
to reduce requirement 59
STORAGE compiler option 85
stream and record files 261, 265
STREAM attribute 245
stream I/O
consecutive data sets 245
data set
access 253
create 249
record format 253
DD statement 250, 254
ENVIRONMENT options 246
file
define 245
PRINT file 254
SYSIN and SYSPRINT files 259
record formats for data transmission 231
string
graphic string constant compilation 38
string assignments 337
string descriptors
string descriptors 498, 501
STRINGOFGRAPHIC compiler option 86
structure of global control blocks
writing the initialization procedure 487
writing the message filtering procedure 487
writing the termination procedure 489
SUB control character 221
summary record, SYSADATA 507
symbol records, SYSADATA 511
symbol table 88
SYNTAX option 86
syntax records, SYSADATA 516
syntax, diagrams, how to read xvii
SYS1.PROCLIB (system procedure library) 239
SYSADATA information, counter records 507
SYSADATA information, file records 508
SYSADATA information, introduction 505
SYSADATA information, literal records 508
SYSADATA information, message records 508
SYSADATA information, options record 507
SYSADATA information, ordinal element records 510
SYSADATA information, ordinal type records 509
SYSADATA information, source records 515
SYSADATA information, summary record 507
SYSADATA information, symbol information 509
SYSADATA information, symbol records 511
SYSADATA information, syntax information 514
SYSADATA information, syntax records 516
SYSADATA information, token records 515
SYSCHK default 479, 480
SYSIN 163, 237
sharing with C 378
SYSIN and SYSPRINT files 259
SYSLIB
%INCLUDE 101
preprocessing 164
SYSLIN 163
SYSOUT 355
SYSPARM compiler option 87
SYSPRINT 237
and z/OS UNIX 237
compiler listing written to 164
effect of STDSYS option on 85
required DD statement 162
shared with older PL/I 173
sharing between enclaves 172
sharing with C 377, 378
specifying on the DD option 19
using MSGFILE(SYSPRINT) 173
using with the PUT statement 259
SYSPUNCH 163
system
failure 481
restart after failure 481
SYSTEM compiler options
SYSTEM(CICS) 87
SYSTEM(IMS) 87
SYSTEM(MVS) 87
SYSTEM(OS) 87
SYSTEM(TSO) 87
type of parameter list 87
SYSUT1 compiler data set 164

T

tab control table 257
temporary workfile
SYSUT1 164
terminal
input 259
capital and lowercase letters 262
COPY option of GET statement 265
defining QSAM files 262
end of file 262
format of data 261
stream and record files 261
output 265
format of PRINT file 265
output from PUT EDIT command 267
stream and record files 265
TERMINAL compiler option 88
terminating
compilation 17

termination procedure
compiler user exit 489
example of procedure-specific control block 489
syntax
global 484
specific 489
TEST compiler option
definition 88
TIME parameter 156
TIMESTAMP
saved load module timestamp in PLIDUMP 475
TITLE option
associating standard SYSPRINT file 173
description under z/OS UNIX 213
using 226
TITLE option under OS/390
specifying character string value 207
TITLE option under z/OS UNIX
using files not associated with data sets 214
token records, SYSADATA 515
traceback table
program unit name in 469
trailer label 224
TYPE option under z/OS UNIX 219

U

U compiler message 110
U option of ENVIRONMENT
for record I/O 230
for stream I/O 246, 247
U-format 223
undefined-length records 223
UNDEFINEDFILE condition
BLKSIZE error 232
line size conflict in OPEN 255
raising when opening a file under z/OS UNIX 220
UNDEFINEDFILE condition under OS/390
DD statement error 208
UNDEFINEDFILE condition under z/OS UNIX
using files not associated with data sets 220
UNIT parameter
consecutive data sets 274
unreferenced identifiers 9
UNROLL compiler option 91
updating
ESDS 305
REGIONAL(1) data set 286
relative-record data set 324
UPPERINC compiler suboption 28
USAGE compiler option 92
user exit
compiler 483
customizing
modifying SYSUEXIT 486
structure of global control blocks 484
writing your own compiler exit 487
user exit *(continued)*
functions 484
sort 348
using
arrays as host variable, SQL preprocessor 133
using host variables, SQL preprocessor 132

V

V option of ENVIRONMENT
for record I/O 230
for stream I/O 246, 247
variable-length records
format 222
sort program 365
VB option of ENVIRONMENT
for record I/O 230
for stream I/O 246, 247
VB-format records 222
VBS option of ENVIRONMENT
for stream I/O 246
VOLUME parameter
consecutive data sets 274
volume serial number
direct access volumes 224
regional data sets 287
VS option of ENVIRONMENT
for stream I/O 246
VSAM (virtual storage access method)
data sets
alternate index paths 301
alternate indexes 313
blocking 292
choosing a type 295
defining 291, 302
defining files for 298
dummy data set 296
entry-sequenced 302
key-sequenced and indexed entry-sequenced 306
keys for 294
organization 292
performance options 301
relative record 320
running a program with 291
specifying ENVIRONMENT options 298
using 291
defining files 298
ENV option 298
performance option 301
indexed data set
load statement and options 306
mass sequential insert 312
relative-record data set 321
VSAM option 301
VTOC 224

W

W compiler message 110
WIDECHAR compiler option 93
WINDOW compiler option 93
work data sets for sort 355
WRITABLE compiler option 93

X

XINFO compiler option 95
XML
support in the SAX parser 403, 433
XML compiler option 45, 97
XREF compiler option 97

Z

z/OS UNIX
compile-time options
specifying 160
compiling under 159
DD_DDNAME environment variable 215
export command 215
setting environment variables 237
specifying compile-time options
command line 160
using flags 161
zero value 231

Readers' Comments — We'd Like to Hear from You

Enterprise PL/I for z/OS
Programming Guide
Version 5 Release 1

Publication No. GI13-4536-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Send your comments to the address on the reverse side of this form.

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
GI13-4536-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM
H150/090
555 Bailey Avenue
San Jose, CA
USA 95141-1099

Fold and Tape | **Please do not staple** | Fold and Tape

GI13-4536-00

Cut or Fold
Along Line

Product Number: 5655-PL5

Printed in USA

GI13-4536-00