Programming Guide

Version 8.5
Contents

Tables ........................................ xi

Figures ........................................ xiii

About this book. ................................. xv
What's new? ...................................... xv
How to read the syntax diagrams .......... xv
How to send your comments ................. xvii

Part 1. Introducing PL/I on your workstation ................. 1

Chapter 1. Porting applications between platforms ......... 3
Getting mainframe applications to compile on the workstation ......... 3
Choosing the right compile-time options .......... 4
Language restricted ............... 4
Using the macro facility to help port programs .... 7
Getting mainframe applications to run on the workstation ......... 7
Linking differences ............. 7
Data representations causing runtime differences .... 7
Environment differences affecting portability .... 10
Language elements causing runtime differences .... 11

Part 2. Compiling and linking your program .......... 13

Chapter 2. Compiling your program .............. 17
A short practice exercise .............. 17
The HELLO program .............. 17
Using compile-time options .......... 18
Using the sample programs provided with the product ......... 18
Preparing to compile source programs ......... 18
Program file structure ............... 18
Program file format ............. 20
Setting compile-time environment variables ......... 21
IBM.OPTIONS ....................... 21
IBM.PPINCLUDE ............... 22
IBM.PPMACRO ............... 22
IBM.PPSQL ....................... 22
IBM.PPCICS ....................... 22
IBM.SOURCE ....................... 23
IBM.SYSLIB ....................... 23
IBM.PRINT ....................... 23
IBM.OBJECT ....................... 23
IBM.DECK ....................... 23
INCLUDE ....................... 24
TMP ............... 24
Using the PLI command to invoke the compiler ......... 24
Where to specify compile-time options ......... 24

IBM.OPTIONS and IBM.PPxxx environment variables .......... 25
PLI command .......... 25
%PROCESS statement ...... 25

Chapter 3. Compile-time options ............. 27
Compile-time option descriptions .......... 27
Rules for using compile-time options .......... 30
ADDEXT ....................... 31
AGGREGATE ............... 32
ATTRIBUTES ............... 33
BIFPREC ............... 34
BLANK ....................... 35
CASERULES ............... 36
CHECK ............... 37
CMPAT ....................... 38
CODEPAGE ............... 39
COMPILE ............... 40
COPYRIGHT ............... 41
CURRENCY ............... 42
DBCS ............... 43
DEFAULT ............... 44
DEPRECADE .......... 53
DEPRECATENEXT .......... 55
DLLINIT ............... 56
EXIT ............... 57
EXTRN ............... 58
FLAG ............... 59
FLOATINMATH .......... 60
GONUMBER ............... 61
GRAPHIC ............... 62
IGNORE ............... 63
IMPRECISE ............... 64
INCAFTER ............... 65
INCLUDE ............... 66
INITAUTO ............... 67
INITBASED ............... 68
INITCTL ............... 69
INITSTATIC ............... 70
INSOURCE ............... 71
LANG_LVL ............... 72
LIBS ............... 73
LIMITS ............... 74
LINECOUNT ............... 75
LINEDIR ............... 76
LIST ............... 77
LISTVIEW ............... 78
MACRO ............... 80
MARGIN ............... 81
MARGINS ............... 82
MAXGEN ............... 83
MAXMSG ............... 84
MAXNEST ............... 85
MAXSTM ............... 86
MAXTEMP ............... 87
## Chapter 4. PL/I preprocessors . . . . . . 147

**Include preprocessor** 147
- Examples 147
  - Include preprocessor options environment variable 148
- Macro preprocessor 148
  - Macro preprocessor options 148
  - Macro facility options environment variables 150
- SQL preprocessor 150
  - Programming and compilation considerations 150
  - SQL preprocessor options 151

**SQL preprocessor options environment variable** 158
**SQL preprocessor BIND environment variables** 158
**Coding SQL statements in PL/I applications** 158
**Large Object (LOB) support** 167
**User defined functions sample programs** 168
**Handling SQL error return codes** 169
**SQL compatibility and migration considerations** 170
**CICS support** 170
**Programming and compilation considerations** 170
**CICS preprocessor options** 172
**CICS preprocessor options environment variables** 172
**Coding CICS statements in PL/I applications** 173
**Writing CICS transactions in PL/I** 173
**CICS abends used for PL/I programs** 174
**CICS run-time user exit** 174

## Chapter 5. Compilation output . . . . . . 175

**Using the compiler listing** 175
**Compiler output files** 183

## Chapter 6. Linking your program . . . . . . 185

**Starting the linker** 185
- Statically linking 185
- Linking from the command line 185
- Linking from a make file 186
**Input and output** 187
- Search rules 188
- Specifying directories 188
- Filename defaults 188
- Specifying object files 189
**Using response files** 189
**Specifying executable output type** 190
- Producing an .EXE file 190
- Producing a dynamic link library 190
**Packing executables** 191
**Generating a map file** 191
**Linker return codes** 191

## Chapter 7. Setting linker options . . . . . . 193

**Setting options on the command line** 193
**Setting options in the ILINK environment variable** 193
**Using the linker** 193
**Specifying numeric arguments** 194
**Summary of Windows linker options** 194

**Windows linker options** 195
- /? 195
- /ALIGNADDR 195
- /ALIGNFILE 196
- /BASE 196
- /CODE 196
- /DATA 197
- /DBGPACK, /NODBPACK 197
- /DEBUG, /NODEBUG 197
- /DEFUALTLIBRARYSEARCH 197
- /DLL 198
- /ENTRY 198
- /EXECUTABLE 198
- /NOEXTDICTIONARY 199
### Chapter 13. Defining and using workstation VSAM data sets

- Moving data between the workstation and mainframe
- Workstation VSAM organization
- Creating and accessing workstation VSAM data sets
- Determining which type of workstation VSAM data set you need
- Accessing records in workstation VSAM data sets
- Using keys for workstation VSAM data sets
- Defining files for workstation VSAM data sets
- Specifying options of the PL/I ENVIRONMENT attribute
- Adapting existing programs for workstation VSAM

### Chapter 14. Open Database Connectivity

- Introducing ODBC
- Background
- ODBC Driver Manager
- Choosing embedded SQL or ODBC
- Using the ODBC drivers

### Chapter 15. Using java Dclgen

- Understanding java Dclgen terminology
- PL/I java Dclgen support
- Creating a table declaration and host structure
- Selecting a database
- Selecting a table and generation a PL/I declaration
- Modifying and saving the generated PL/I declaration
- Exiting java Dclgen
- Including data declarations in your program

### Part 5. Using PL/I with databases

- Defining files for a regional data set
- Specifying ENVIRONMENT options
- Essential information for creating and accessing regional data sets
- Using keys with regional data sets
- Using REGIONAL(1) data sets
- Dummy records
- Creating a REGIONAL(1) data set
- Example
- Accessing and updating a REGIONAL(1) data set
- Sequential access
- Direct access
- Example

### Part 6. Advanced topics

- Online help
- Environment-specific information
- Connecting to a data source
- Error messages
- ODBC APIs from PL/I
- CALL interface convention
- Using the supplied include files
- Mapping of ODBC C types
- Setting licensing information for ODBC Driver Manager
- Sample program using supplied include files

- Chapter 16. Using the Program Maintenance Utility, NMAKE

- Why use NMAKE?
- Running NMAKE
- Using the command line
- Using NMAKE command files
- NMAKE options
- Produce error file (/X)
- Build all targets (/A)
- Suppress messages (/C)
- Display modification dates (/D)
- Override environment variables (/E)
- Specify description file (/F)
- Display help (/HELP or /?)
- Ignore exit codes (/I)
- Display commands (/N)
- Suppress sign-on banner (/NOLOGO)
- Print macro and target definitions (/P)
- Return exit code (/Q)
- Ignore TOOLS.INI file (/R)
- Suppress command display (/S)
- Change target modification dates (/T)
- Description files
- Description blocks
- Special features
- Targets in several description blocks
- Using macros
- Macros example
- Special features
- Macros in a description file
- Macros on the command line
- Inherited macros
- Defined macros
- Macro substitutions
### Contents

**Chapter 17. Improving performance** 355
Selecting compile-time options for optimal performance.
- OPTIMIZE 355
- IMPRECISE 355
- GONUMBER 356
- SNAP 356
- RULES 356
- PREFIX 357
- DEFAULT 357
- Summary of compile-time options that improve performance. 361
Coding for better performance 361
- DATA-directed input and output 361
- Input-only parameters 362
- String assignments 362
- Loop control variables 362
- PACKAGES versus nested PROCEDUREs 363
- REDUCIBLE functions 364
- DEFINED versus UNION 364
- Named constants versus static variables 365
- Avoiding calls to library routines. 366

**Chapter 18. Using user exits** 369
Using the compiler user exit 369
- Procedures performed by the compiler user exit 369
- Activating the compiler user exit 370
- The IBM-supplied compiler exit, IBMUEXIT 370
- Customizing the compiler user exit 371
- Using the CICS run-time user exit 375
- Prior to program invocation 376
- After program termination 376
- Modifying CEEFXTA 376
- Using data conversion tables 376

**Chapter 19. Building dynamic link libraries** 379
Creating DLL source files 379
- Compiling your DLL source 379
Preparing to link your DLL.
- Specifying exported names under Windows 380
- Linking your DLL 380
- Using your DLL 380
Sample program to build a DLL 381
- Using FETCH and RELEASE in your main program . 381
- Exporting data from a DLL. 382

**Chapter 20. Using IBM Library Manager on Windows** 383
Running ILIB 383
- Using the command line 383
- Using the ILIB environment variable 384
- Using an ILIB response file 385
- Examples specifying ILIB parameters 385
Controlling ILIB input 386
- Controlling ILIB output. 386
- Controlling ILIB output. 387
ILIB objects 388
- Summary of ILIB objects 388
- Add/Replace 388
- /EXTRACT 389
- /REMOVE 390
ILIB options 390
- Summary of ILIB options 390
- /? 391
- /BACKUP 391
- /DEF 391
- /FREEFORMAT 392
- /GENDEF 392
- /GL 392
- /HELP 392
- /LIST 393
- /NOEXT 393
- /OUT 393
- /QUIET 393
- /WARN 393

**Chapter 21. Calling conventions** 395
Understanding linkage considerations 395
- OPTLINK linkage 396
- Features of OPTLINK 396
- Tips for using OPTLINK 397
- General-purpose register implications 397
- Parameters 397
- Examples of passing parameters 398
SYSTEM linkage 403
- Features of SYSTEM 403
- Example using SYSTEM linkage 404
STDSCALL linkage (Windows only) 405
<table>
<thead>
<tr>
<th>Features of STDCALL</th>
<th>406</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples using the STDCALL convention</td>
<td>406</td>
</tr>
<tr>
<td>Using WinMain (Windows only)</td>
<td>408</td>
</tr>
<tr>
<td>CDECL linkage</td>
<td>408</td>
</tr>
<tr>
<td>Features of CDECL</td>
<td>408</td>
</tr>
<tr>
<td>Examples using the CDECL convention</td>
<td>409</td>
</tr>
</tbody>
</table>

**Chapter 22. Using PL/I in mixed-language applications** | 411 |
| Matching data and linkages | 411 |
| What data is passed | 411 |
| How data is passed | 413 |
| Where data is passed | 415 |
| Maintaining your environment | 415 |
| Invoking non-PL/I routines from a PL/I MAIN | 415 |
| Invoking PL/I routines from a non-PL/I main | 416 |
| Using ON ANYCONDITION | 416 |

**Chapter 23. Interfacing with Java** | 419 |
| What is the Java Native Interface (JNI)? | 419 |
| JNI Sample Program #1 - 'Hello World' | 420 |
| Step 1: Writing the Java Program | 420 |
| Step 2: Compiling the Java Program | 421 |
| Step 3: Writing the PL/I Program | 421 |
| Step 4: Compiling and Linking the PL/I Program | 422 |
| Step 5: Running the Sample Program | 423 |
| JNI Sample Program #2 - Passing a String | 423 |
| Step 1: Writing the Java Program | 423 |
| Step 2: Compiling the Java Program | 424 |
| Step 3: Writing the PL/I Program | 425 |
| Step 4: Compiling and Linking the PL/I Program | 426 |
| Step 5: Running the Sample Program | 427 |
| JNI Sample Program #3 - Passing an Integer | 427 |
| Step 1: Writing the Java Program | 427 |
| Step 2: Compiling the Java Program | 429 |
| Step 3: Writing the PL/I Program | 429 |
| Step 4: Compiling and Linking the PL/I Program | 430 |
| Step 5: Running the Sample Program | 431 |
| Determining equivalent Java and PL/I data types | 431 |

**Chapter 24. Using sort routines** | 433 |
| Comparing S/390 and workstation sort programs | 433 |
| Preparing to use sort | 434 |
| Choosing the type of sort | 435 |
| Specifying the sorting field | 437 |
| Specifying the records to be sorted | 438 |
| Calling the sort program | 438 |
| PLISRT examples | 438 |
| Determining whether the sort was successful | 439 |
| Getting diagnostic information | 440 |
| The IBM_SORT_RC environment variable | 440 |
| The IBM_SORT_DIR environment variable | 441 |
| The IBM_SORT_MSGFILE environment variable | 441 |
| Sort data input and output | 441 |
| Sort data handling routines | 441 |
| E15 — input-handling routine (sort exit E15) | 442 |
| E35 — output-handling routine (sort exit E35) | 444 |
| Calling PLISRTA | 446 |
| Calling PLISRTB | 447 |
| Calling PLISRTOC | 448 |
| Calling PLISRTD, example 1 | 449 |
| Calling PLISRTD, example 2 | 451 |

**Chapter 25. Using the SAX parser** | 453 |
| Overview | 453 |
| The PLISAXA built-in subroutine | 454 |
| The PLISAXB built-in subroutine | 454 |
| The SAX event structure | 454 |
| start_of_document | 455 |
| version_information | 455 |
| encoding_declaration | 455 |
| standalone_declaration | 455 |
| document_type_declaration | 455 |
| end_of_document | 455 |
| start_of_element | 455 |
| attribute_name | 456 |
| attribute_characters | 456 |
| attribute_predefined_reference | 456 |
| attribute_character_reference | 456 |
| end_of_element | 456 |
| start_of_CDATA_section | 456 |
| end_of_CDATA_section | 456 |
| content_characters | 457 |
| content_predefined_reference | 457 |
| content_character_reference | 457 |
| processing_instruction | 458 |
| comment | 458 |
| unknown_attribute_reference | 458 |
| unknown_content_reference | 458 |
| start_of_prefix_mapping | 458 |
| end_of_prefix_mapping | 458 |
| exception | 458 |
| Parameters to the event functions | 458 |
| Coded character sets for XML documents | 459 |
| Supported EBCDIC code pages | 459 |
| Supported ASCII code pages | 460 |
| Specifying the code page | 460 |
| Exceptions | 461 |
| Example | 462 |
| Continuable exception codes | 473 |
| Terminating exception codes | 477 |

**Chapter 26. Using PL/I MLE in your applications** | 481 |
| Applying attributes and options | 481 |
| DATE attribute | 481 |
| RESPECT compile-time option | 482 |
| WINDOW compile-time option | 482 |
| RULES compile-time option | 482 |
| Understanding date patterns | 483 |
| Patterns and windowing | 483 |
| Using built-in functions with MLE | 484 |
| DAYS | 484 |
| DAYSTODATE | 485 |
| Performing date calculations and comparisons | 485 |
| Explicit date calculations | 485 |
| Implicit date calculations | 486 |
Implicit date comparisons ........ 486
Implicit DATE assignments .......... 487

Part 7. Appendixes ................. 489

Notices .................................. 491
Programming interface information  492
Macros for customer use ............. 492
Trademarks ............................... 492

Bibliography............................ 495

PL/I publications ...................... 495
PL/I for AIX ............................. 495
Enterprise PL/I publications ........ 495
Related publications .................. 495
DB2 UDB for OS/390 and z/OS ........ 495
TXSeries for Multiplatforms ......... 495

Glossary ................................. 497

Index ................................ 515
### Tables

1. Normalized IEEE versus normalized hexadecimal ........................................... 9
2. Compile-time options, abbreviations, and IBM-supplied defaults ....................... 27
3. SQL data types generated from PL/I declarations ............................................. 164
4. SQL data types mapped to PL/I declarations ................................................. 164
5. Aggregate length table headings and description ........................................... 181
6. Possible compilation disk outputs ................................................................. 183
7. Linker filename defaults ................................................................................. 188
8. Windows linker options summary ................................................................. 194
9. Statements, attributes, and options for native data sets ................................... 251
10. Statements, attributes, and options for workstation VSAM data sets ................. 252
11. ANSI print control characters ....................................................................... 255
12. IBM Proprinter equivalents to ANSI control characters .................................... 256
13. Statements and options allowed for creating and accessing consecutive data sets ......................................................... 270
14. Statements and options allowed for creating and accessing regional data sets ......................................................... 277
15. Types and advantages of workstation VSAM data sets ................................... 289
16. Processing allowed on alternate index paths .................................................. 292
17. Statements and options allowed for loading and accessing workstation VSAM sequential data sets ......................................................... 295
18. Statements and options allowed for loading and accessing workstation VSAM keyed data sets ......................................................... 298
19. Statements and options allowed for loading and accessing workstation VSAM direct data sets ......................................................... 305
20. Supplied include files for ODBC .................................................................. 319
21. ODBC names truncated or abbreviated for PL/I ........................................... 319
22. Mapping of ODBC C Type to PL/I Data Declarations ...................................... 320
23. Declarations generated by java Delgen .......................................................... 324
24. NMAKE Predefined Inference Rules .............................................................. 347
25. Conditions under which conversions are handled inline .................................. 366
26. Conditions under which string built-in functions are handled inline ................ 367
27. ILIB objects on Windows ............................................................................... 388
28. ILIB options on Windows .............................................................................. 390
29. Equivalent data types between C and PL/I .................................................... 412
30. Java Primitive Types and PL/I Native Equivalents ......................................... 431
31. workstation PLISRTx .................................................................................. 433
32. Return codes from PLISRTx .......................................................................... 440
33. Terminating Exceptions .................................................................................. 473
34. Continuable Exceptions .................................................................................. 477
35. Date patterns supported by PL/I MLE ............................................................ 483
# Figures

1. The PL/I declaration of SQLCA .......... 159
2. The PL/I declaration of an SQL descriptor area .......... 160
3. CHIMES program compiler listing .......... 176
4. Make file example .......... 187
5. PL/I code that produces a formatted dump .......... 217
6. Example of PLIDUMP output .......... 218
7. Static and dynamic descendant procedures .......... 220
8. Creating a data set with stream-oriented data transmission .......... 259
9. Accessing a data set with stream-oriented data transmission .......... 261
10. Creating a print file via stream data transmission .......... 263
11. Declaration of PLITABS .......... 264
12. PL/I structure PLITABS for modifying the preset tab settings .......... 265
13. A sample interactive program .......... 269
14. Merge Sort—Creating and accessing a consecutive data set .......... 273
15. Printing record-oriented data transmission .......... 275
16. Creating a REGIONAL(1) data set .......... 281
17. Updating a REGIONAL(1) data set .......... 284
18. VSAM data sets and allowed file attributes .......... 291
19. Creating a workstation VSAM keyed data set .......... 294
20. Defining and loading a workstation VSAM sequential data set .......... 297
21. Defining and loading a workstation VSAM keyed data set .......... 301
22. Updating a workstation VSAM keyed data set .......... 303
23. Loading a workstation VSAM direct data set .......... 308
24. Updating a workstation VSAM direct data set by key .......... 310
25. Selecting a database .......... 325
26. Display of tables created by the qualifier .......... 326
27. Generated PL/I declarations .......... 327
28. PL/I compiler user exit procedures .......... 370
29. Example of an IBMUEXIT.INF file .......... 371
30. Java Sample Program #2 - Passing a String .......... 424
31. PL/I Sample Program #2 - Passing a String .......... 426
32. Java Sample Program #3 - Passing an Integer .......... 428
33. PL/I Sample Program #3 - Passing an Integer .......... 430
34. Flow of control for the sort program .......... 436
35. Skeletal code for an input procedure .......... 443
36. When E15 is external to the procedure calling PLISRTx .......... 444
37. Skeletal code for an output-handling procedure .......... 445
38. PLISRTA—Sorting from input data set to output data set .......... 446
39. PLISRTB—Sorting from input-handling routine to output data set .......... 447
40. PLISRTC—Sorting from input data set to output-handling routine .......... 448
41. PLISRTD—Sorting input-handling routine to output-handling routine .......... 449
42. PLISRTD—Sorting input-handling routine to output-handling routine .......... 451
43. Sample XML document .......... 455
44. PLISAXA coding example - type declarations .......... 462
45. PLISAXA coding example - event structure .......... 463
46. PLISAXA coding example - main routine .......... 464
47. PLISAXA coding example - event routines .......... 465
48. PLISAXA coding example - program output .......... 473
About this book

This Programming Guide is designed to help use the PL/I for Windows compilers to code and compile PL/I programs.

If you have typically used mainframe PL/I and are interested in moving your programs to the Windows platform, Chapter 1, “Porting applications between platforms,” on page 3 should be particularly useful. Other information in this guide will help you understand some basic Windows features as well as give instructions on how to compile, link, and run a PL/I program.

What's new?

Some of the most recent additions to the PL/I workstation compilers include:

- Some tips on how to use PL/I and Java together
- How to use dclgen in the Windows environment
- Help with using the Open Database Connectivity

How to read the syntax diagrams

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

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.

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. 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.
How to read syntax diagrams

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

```
  ▶️REQUIRED_ITEM ▶️
```

**Optional Items**
Optional items appear below the main path.

```
  ▶️REQUIRED_ITEM ◀️
  OPTIONAL_ITEM
```

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.

```
  ▶️REQUIRED_ITEM ◀️
  OPTIONAL_ITEM
```

**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.

```
  ▶️REQUIRED_ITEM ◀️
  ▶️required_choice1
  ▶️required_choice2
```

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

```
  ▶️REQUIRED_ITEM ◀️
  ▶️optional_choice1
  ▶️optional_choice2
```

**Repeatable items**
An arrow returning to the left above the main line indicates that an item can be repeated.
How to read syntax diagrams

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**

Default keywords appear above the main path, and the remaining choices are shown below the main path.

**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.

---

**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/
How to send your comments

Be sure 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.
Part 1. Introducing PL/I on your workstation

Chapter 1. Porting applications between platforms

Getting mainframe applications to compile on the workstation 3
- Choosing the right compile-time options 4
- Language restricted 4
  - RECORD I/O 4
  - STREAM I/O 4
- Structure expressions 5
- Array expressions 5
- DEFAULT statement 5
- Extents of automatic variables 5
- Built-in functions 5
- iSUB defining 6
- DBCS 6
- Macro preprocessor 6
- Using the macro facility to help port programs 7

Getting mainframe applications to run on the workstation 7
- Linking differences 7
- Data representations causing runtime differences 7
- Environment differences affecting portability 10
- Language elements causing runtime differences 11
Chapter 1. Porting applications between platforms

The IBM mainframe environment has a different hardware and operating system architecture than your AIX system or your personal computer (PC). Operating systems other than the mainframe are sometimes referred to as workstation platforms. In this book, we use the term workstation to refer to the AIX and Windows operating systems.

Some problems can arise as you move PL/I programs between the mainframe and workstation environments. The problems may be caused by fundamental platform differences or differences between the PL/I for Windows compiler and the Enterprise PL/I for z/OS compiler. This chapter describes some of these differences between development platforms, and then provides instructions that minimize problems in the following areas:

• Compiling mainframe applications without error on the workstation.
• Running mainframe applications on the workstation (and getting the same results).
• Writing, compiling, and testing applications on the workstation that are later run in production mode on the mainframe.

Getting mainframe applications to compile on the workstation

As you move programs to your workstation from the mainframe, one of your first goals is to get the applications you have already been using to compile in the new environment without errors.

The character sets used on the mainframe and workstation are different and can cause some compile problems:

Embedded control characters
If a source file contains characters with hexadecimal values less than '20'x, the workstation compiler might misinterpret the size of a line in that file, or even the size of the file itself. You should use hexadecimal character constants to encode these values.

If you are downloading a source file from the host that has variables initialized to values entered with a hexadecimal editor, some of those values might have a hexadecimal value less than '20'x even though they have greater values on the host.

National characters and other symbols
Transferring programs between platforms can cause errors if you use national characters and other symbols (in PL/I context) in certain code pages. This is true of the logical “not” (¬) and “or” (1) signs, the currency symbol, and use of the following alphabetic extenders in PL/I identifiers:

$ 
# 
@

To avoid potential problems involving “not”, “or” and the currency symbol, use the NOT (see “NOT” on page 94), OR (see “OR” on page 101), and CURRENCY (see “CURRENCY” on page 42) compile time options on the...
Getting mainframe applications to compile on the workstation

*PROCESS statement. Avoid potential problems involving other characters by using the NAMES (see “NAMES” on page 91) compile-time option to define extramural characters and symbols.

Choosing the right compile-time options

By selecting certain compile time options, you can make your source code more portable across compilers and platforms.

If you want to improve compatibility with pre-Enterprise PL/I, you could specify the following options:

- DEFAULT( DESCLOCATOR EVENDEC NULL370 RETURNS(BYADDR) )
- LIMITS( EXTNAM(7) NAME(31) )

Note that the option DEFAULT(RETURNS(BYADDR)) will make the invocation of a non-PL/I function on the workstation fail unless the BYVALUE attribute is specified in the RETURNS description.

These (and all the other compiler) options are listed alphabetically in Chapter 3, “Compile-time options,” on page 27, where they are also described in detail.

Language restricted

Except where indicated, the compiler will flag the use of any language that is restricted.

RECORD I/O

RECORD I/O is supported, but with the following restrictions:

- The EVENT clause on READ/WRITE statements is not supported.
- The UNLOCK statement is not supported.
- The following options of the ENVIRONMENT attribute are not supported, but their use is flagged only under LANGLVL(NOEXT):
  - ADDBUFF
  - ASCII
  - BUFFERS
  - BUFND
  - BUFINI
  - BUFOFF
  - INDEXAREA
  - LEAVE
  - NCP
  - NOWRITE
  - REGIONAL(2)
  - REGIONAL(3)
  - REREAD
  - SIS
  - SKIP
  - TOTAL
  - TP
  - TRKOFL

STREAM I/O

STREAM I/O is supported, but the following restrictions apply to PUT/GET DATA statements:

- DEFINED is not supported if the DEFINED variable is BIT or GRAPHIC or has a POSITION attribute.
Getting mainframe applications to compile on the workstation

- DEFINED is not supported if its base variable is an array slice or an array with a different number of dimensions than the defined variable.

**Structure expressions**
Structure expressions as arguments are not supported unless both of the following conditions are true:
- There is a parameter description.
- The parameter description specifies all constant extents.

**Array expressions**
An array expression is not allowed as an argument to a user function unless it is an array of scalars of known size. Consequently, any array of scalars of arithmetic type may be passed to a user function, but there may be problems with arrays of varying-length strings.

The following example shows a numeric array expression supported in a call:

```plaintext
dcl x entry, (y(10), z(10)) fixed bin(31);
call x(y + z);
```

The following unprototyped call would be flagged since it requires a string expression of unknown size:

```plaintext
dcl a1 entry;
dcl (b(10), c(10)) char(20) var;
call a1(b || c);
```

However, the following prototyped call would not be flagged:

```plaintext
dcl a2 entry(char(30) var);
dcl (b(10), c(10)) char(20) var;
call a2(b || c);
```

**DEFAULT statement**
Factored default specifications are not supported.

For example, a statement such as the following is not supported:

```plaintext
default ( range(a:h), range(p:z) ) fixed bin;
```

But you could change the above statement to the following equivalent and supported statement:

```plaintext
default range(a:h) fixed bin, range(p:z) fixed bin;
```

The use of a "(" after the DEFAULT keyword is reserved for the same purpose as under the ANSI standard: after the DEFAULT keyword, the standard allows a parenthesized logical predicate in attributes.

**Extents of automatic variables**
An extent of an automatic variable cannot be set by a function nested in the procedure where the automatic variable is declared or by an entry variable unless the entry variable is declared before the automatic variable.

**Built-in functions**
Built-in functions are supported with the following exceptions/restrictions:
- The PLITEST built-in function is not supported.
- Pseudovariables are not supported in:
Getting mainframe applications to compile on the workstation

- The STRING option of PUT statements

- Pseudovariables permitted in DO loops are restricted to:
  - IMAG
  - REAL
  - SUBSTR
  - UNSPEC

- The POLY built-in function has the following restrictions:
  - The first argument must be REAL FLOAT.
  - The second argument must be scalar.

- The COMPLEX pseudovariable is not supported.

- The IMS built-in subroutines PLICANC, PLICKPT, and PLIREST are not supported.

**iSUB defining**
Support for iSUB defining is limited to arrays of scalars.

**DBCS**
DBCS can be used only in the following:
- G and M constants
- Identifiers
- Comments

G literals can start and end with a DBCS quote followed by either a DBCS G or an SBCS G.

**Macro preprocessor**
Suffixes that follow string constants are not replaced by the macro preprocessor—whether or not these are legal PL/I suffixes—unless you insert a delimiter between the ending quotation mark of the string and the first letter of the suffix.

OS PL/I V2R1 compiler introduced this change. As an example, consider:

```pli
%DCL (GX, XX) CHAR;
%GX='||FX';
%XX='||ZZ';
DATA = 'STRING'GX;
DATA = 'STRING'XX;
DATA = 'STRING' GX;
DATA = 'STRING' XX;
```

Under OS PL/I V1, this produces the source:

```pli
DATA = 'STRING'||FX;
DATA = 'STRING'||ZZ;
DATA = 'STRING' ||FX;
DATA = 'STRING' ||ZZ;
```

However, under PL/I for MVS & VM and Enterprise PL/I for z/OS, it produces the source:

```pli
DATA = 'STRING'GX;
DATA = 'STRING'XX;
DATA = 'STRING' ||FX;
DATA = 'STRING' ||ZZ;
```
Using the macro facility to help port programs

In many cases, potential portability problems can be avoided by using the macro facility because it has the capability of isolating platform-specific code. For example, you can include platform-specific code in a compilation for a given platform and exclude it from compilation for a different platform.

The PL/I for Windows macro facility COMPILETIME built-in function returns the date using the format 'DD.MMM.YY', while the PL/I for z/OS macro facility COMPILETIME built-in function returns the date using the format 'DD MMM YY'.

This allows you to write code that can contain conditional system-dependent code that compiles correctly under PL/I for Windows and all versions of the mainframe PL/I compiler, for example:

```pli
%dcl compiletime builtin;
%if substr(compiletime,3,1) = '.' %then
    %do;
    /* Windows PL/I code */
    %end;
%else
    %do;
    /* z/OS PL/I code */
    %end;
```

For information about the macro facility, see the PL/I Language Reference.

Getting mainframe applications to compile on the workstation

Once you have downloaded your source program from the mainframe and compiled it using the workstation compiler without errors, the next step is to run the program. If you want to get the same results on the workstation as you do on the mainframe, you need to know about elements and behavior of the PL/I language that vary due to the underlying hardware or software architecture.

Linking differences

Every .EXE that you build must contain exactly one main routine, that is, exactly one procedure containing OPTIONS(MAIN). If no main routine exists, the linker complains that your program has no starting address. If more than one main routine exists, the linker complains that there are duplicate references to the name main.

Every .DLL that you build must have at least one module compiled with the DLLINIT compile-time option (see “DLLINIT” on page 56).

Data representations causing runtime differences

Most programs act the same without regard to data representation, but to ensure that this is true for your programs, you need to understand the differences described in the following sections.

The workstation compilers support options that instruct the operating system to treat data and floating-point operations the same way that the mainframe does. There are suboptions of the DEFAULT compile-time option that you should specify for all mainframe applications that might need to be changed when moving code to the workstation:

- DEFAULT(EBCDIC) instead of ASCII
- DEFAULT(HEXADEC) instead of IEEE
Getting mainframe applications to run on the workstation

- DEFAULT(E(HEXADEC)) instead of DFT(E(IEEE))
- DEFAULT (NONNATIVE) instead of NATIVE
- DEFAULT (NONNATIVEADDR) instead of NATIVEADDR

For more information on these compile-time options, see “DEFAULT” on page 44.

ASCII versus EBCDIC

Workstation operating systems use the ASCII character set while the mainframe uses the EBCDIC character set. This means that most characters have a different hexadecimal value. For example, the hexadecimal value for a blank is ‘20’x in the ASCII character set and ‘40’x in the EBCDIC character set.

This means that code dependent on the EBCDIC hexadecimal values of character data can logically fail when run using ASCII. For example, code that tests whether or not a character is a blank by comparing it with ‘40’x fails when run using ASCII. Similarly, code that changes letters to uppercase by using ‘OR’ and ‘80’b4 fails when run using ASCII. (Code that uses the TRANSLATE built-in function to change to uppercase letters, however, does not fail.)

In the ASCII character set, digits have the hexadecimal values ‘30’x through ‘39’x. The ASCII lowercase letter ‘a’ has the hexadecimal value ‘61’x, and the uppercase letter ‘A’ has the hexadecimal value ‘41’x. In the EBCDIC character set, digits have the hexadecimal values ‘F0’x through ‘F9’x. In EBCDIC, the lowercase letter ‘a’ has the hexadecimal value ‘81’x, and the uppercase letter ‘A’ has the hexadecimal value ‘C1’x. These differences have some interesting consequences:

- While ‘a’ < ‘A’ is true for EBCDIC, it is false for ASCII.
- While ‘A’ < ‘1’ is true for EBCDIC, it is false for ASCII.
- While x >= ‘0’ almost always means that x is a digit in EBCDIC, this is not true for ASCII.

Because of the differences described, the results of sorting character strings are different under EBCDIC and ASCII. For many programs, this has no effect, but you should be aware of potential logic errors if your program depends on the exact sequence in which some character strings are sorted.

For information on converting from ASCII to EBCDIC, see “Using data conversion tables” on page 376.

NATIVE versus NONNATIVE

The personal computer (PC) holds integers in a form that is byte-reversed when compared to the form in which they are held on the mainframe or AIX. This means, for example, that a FIXED BIN(15) variable holding the value 258, which equals 256+2, is held in storage on Windows as ‘0201’x and on AIX or the mainframe as ‘0102’x. A FIXED BIN(31) variable with the same value would be held as ‘02010000’x on Windows and as ‘00000102’x on AIX or the mainframe.

The AIX and mainframe representation is known as Big Endian (Big End In).

The Windows representation is known, conversely, as Little Endian (Little End In).

This difference in internal representations affects:
- FIXED BIN variables requiring 2 or more bytes
- OFFSET variables
- The length prefix of VARYING strings

8 PL/I for Windows Programming Guide
Getting mainframe applications to run on the workstation

- Ordinal and area data

For most programs, this difference should not create any problems. If your program depends on the hexadecimal value of an integer, however, you should be aware of potential logic errors. Such a dependency might exist if you use the UNSPEC built-in function with a FIXED BINARY argument, or if a BIT variable is based on the address of a FIXED BINARY variable.

If your program manipulates pointers as if they were integers, the difference in data representation can cause problems. If you specify DEFAULT(NONNATIVE), you probably also need to specify DEFAULT(NONNATIVEADDR).

You can specify the NONNATIVE attribute on selected declarations. For example, the assignment in the following statement converts all the FIXED BIN values in the structure from nonnative to native:

```pli
   dcl
   1 a1 native,
       2 b fixed bin(31),
       2 c fixed dec(8,4),
       2 d fixed bin(31),
       2 e bit(32),
       2 f fixed bin(31);
   dcl
   1 a2 nonnative,
       2 b fixed bin(31),
       2 c fixed dec(8,4),
       2 d fixed bin(31),
       2 e bit(32),
       2 f fixed bin(31);
   a1 = a2;
```

IEEE versus HEXADEC

Workstation operating systems represent floating-point data using the IEEE format while the mainframe traditionally uses the hexadecimal format. Table 1 summarizes the differences between normalized floating-point IEEE and hexadecimal:

Table 1: Normalized IEEE versus normalized hexadecimal

<table>
<thead>
<tr>
<th>Specification</th>
<th>IEEE (AIX)</th>
<th>IEEE (PC)</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approximate range of values</td>
<td>±10E-308 to 10E+308</td>
<td>±3.30E-4932 to 1.21E+4932</td>
<td>±10E-78 to 10E+75</td>
</tr>
<tr>
<td>Maximum precision for FLOAT DECIMAL</td>
<td>32</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>Maximum precision for FLOAT BINARY</td>
<td>106</td>
<td>64</td>
<td>109</td>
</tr>
<tr>
<td>Maximum number of digits in FLOAT DECIMAL exponent</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Maximum number of digits in FLOAT BINARY exponent</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Hexadecimal float has the same maximum and minimum exponent values for short, long, and extended floating-point, but IEEE float has differing maximum and minimum exponent values for short, long, and extended floating-point. This means that while 1E74, which in PL/1 should have the attributes FLOAT DEC(1), is a valid hexadecimal short float, it is not a valid IEEE short float.
Getting mainframe applications to run on the workstation

For most programs these differences should create no problems, just as the different representations of FIXED BIN variables should create no problems. However, use caution in coding if your program depends on the hexadecimal value of a float value.

Also, while FIXED BIN calculations produce the same result independent of the internal representations described above, floating-point calculations do not necessarily produce the same result. When the compiler converts a floating-point literal into its internal binary representation, the representations might not be the same as those from different platforms. The results from the floating-point calculation on different platforms might also be slightly different. This is particularly true for short and extended floating-point.

The differences are usually very small and might cause rounding errors. Programs that depend on floating-point values being exact might get incorrect results. For example, if a file contains a floating-point binary value, you might search the file for that value and fail to find an exact match.

**EBCDIC DBCS versus ASCII DBCS**

EBCDIC DBCS strings are enclosed in shift codes, while ASCII DBCS strings are not enclosed in shift codes. The hexadecimal values used to represent the same characters are also different.

Again, for most programs this should make no difference. If your program depends on the hexadecimal value of a graphic string or on a character string containing mixed character and graphic data, use caution in your coding practices.

**Environment differences affecting portability**

There are some differences, other than data representation, between the workstation and mainframe platforms that can also affect the portability of your programs. This section describes some of these differences.

**File names**

File naming conventions on the PC are very different from those on the mainframe. The following file name, for example, is valid on the PC, but not on the mainframe:

d:\programs\data\myfile.dat

This can affect portability if you use file names in your PL/I source as part of the TITLE option of the OPEN and FETCH statements.

**File attributes**

PL/I allows many file attributes to be specified as part of the ENVIRONMENT attribute in a file declaration. Many of these attributes have no meaning on the workstation, in which case the compiler ignores them. If your program depends on these attributes being respected, your program is not likely to port successfully.

**Control codes**

Some characters that have no particular meaning on the mainframe are interpreted as control characters by the workstation and can lead to incorrect processing of data files having a TYPE of either LF, LFEOF, CRLF, or CRLFEOF. Such files should not contain any of the following characters:

- 0A’x (“LF - line feed”)
- 0D’x (“CR - carriage return”)
- 1A’x (“EOF - end of file”)

- 3A’x (“SI - start of information”)
Getting mainframe applications to run on the workstation

For example, if the file in the code below has TYPE(CRLF), the WRITE statement raises the ERROR condition with oncode 1041 because 2573 has the hexadecimal value '0D0A'x. This would not occur if the file had TYPE of either FIXED, VARLS, or VARMS.

dcl
  1 a native,
  2 b char(10),
  2 c fixed bin(15),
  2 d char(10);

dcl f file output;

  a.b = 'alpha';
  a.c = 2573;
  a.d = 'omega';

  write file(f) from(a);

Device-dependent control codes
Use of device-dependent (platform-specific) control codes in your programs or files can cause problems when trying to port them to other platforms that do not necessarily support the control codes.

As with all other very platform-specific code, it is best to isolate such code as much as possible so that it can be replaced easily when you move the application to another platform.

Language elements causing runtime differences

The PL/I compiler implements some language elements differently on Windows and z/OS. These differences in implementation might cause differences in the way your program runs. Each of the following items is described in terms of its behavior on Windows.

**FIXED BIN(p) maps to 1 byte if p <= 7**

If you have any variables declared as FIXED BIN with a precision of 7 or less, they occupy 1 byte of storage under PL/I for Windows instead of two as under PL/I for z/OS. If the variable is part of a structure, this usually changes how the structure is mapped, and that could affect how your program runs. For example, if the structure were read in from a file created on the mainframe, fewer bytes would be read in.

To avoid this difference, you could change the precision of the variable to a value between 8 and 15 (inclusive).

**INITIAL attribute for AREAs is ignored**

To keep PL/I for Windows from ignoring the INITIAL attribute for AREAs, convert INITIAL clauses into assignment statements.

For example, in the following code fragment, the elements of the array are not initialized to a1, a2, a3, and a4.

dcl (a1,a2,a3,a4) area;
dcl a(4) area init( a1, a2, a3, a4 );

However, you can rewrite the code as follows so that the array is initialized as wanted.

dcl (a1,a2,a3,a4) area;
dcl a(4) area;

  a(1) = a1;
  a(2) = a2;
  a(3) = a3;
  a(4) = a4;
Issuing of ERROR messages

When the ERROR condition is raised, no ERROR message is issued under PL/I for Windows if the following two conditions are met:

- There is an ERROR ON-unit established.
- The ERROR ON-unit recovers from the condition by using a GOTO to transfer control out of the block.

ERROR messages are directed to STDERR rather than to the SYSPRINT data set. By default, this is the terminal. If SYSPRINT is directed to the terminal, any output in the SYSPRINT buffer (not yet written to SYSPRINT) is written before any ERROR message is written.

ADD, DIVIDE, and MULTIPLY do not return scaled FIXED BIN

Under the RULES(IBM) compile-time option, which is the default, variables can be declared as FIXED BIN with a nonzero scale factor. Infix, prefix, and comparison operations are performed on scaled FIXED BIN as with the mainframe. However, when the ADD, DIVIDE, or MULTIPLY built-in functions have arguments with nonzero factors or specify a result with a nonzero scale factor, the PL/I for Windows compilers evaluate the built-in function as FIXED DEC rather than FIXED BIN as the mainframe compiler.

For example, the PL/I for Windows compilers would evaluate the DIVIDE built-in function in the assignment statement below as a FIXED DEC expression:

```pli
dcl (i,j) fixed bin(15);
dcl x fixed bin(15,2);
.
.
.
x = divide(i,j,15,2)
```

Enablement of OVERFLOW and ZERODIVIDE

For OVERFLOW and ZERODIVIDE, the ERROR condition is raised under the following conditions:

- OVERFLOW or ZERODIVIDE is raised and the corresponding ON-unit is entered.
- Control does not leave the ON-unit through a GOTO statement.
Part 2. Compiling and linking your program

Chapter 2. Compiling your program
A short practice exercise
The HELLO program
Using compile-time options
Using the sample programs provided with the product
Preparing to compile source programs
Program file structure
Using a PROCEDURE statement with PROCESS:
INCLUDE processing
%LINE directive
Margins
Program file format
Line continuation
Setting compile-time environment variables
IBM.OPTIONS
IBM.PPINCLUDE
IBM.PPMACRO
IBM.PPCICS
IBM.SOURCE
IBM.SYSLIB
IBM.PRINT
IBM.OBJECT
IBM.DECK
INCLUDE
TMP
Using the PLI command to invoke the compiler
Where to specify compile-time options
IBM.OPTIONS and IBM.PPxxx environment variables
PLI command
%PROCESS statement

Chapter 3. Compile-time options
Compile-time option descriptions
Rules for using compile-time options
ADDEXT
AGGREGATE
ATTRIBUTES
BIPREC
BLANK
CASERULES
CHECK
CMAT
CODEPAGE
COMPILE
COPYRIGHT
CURRENCY
DBC
DEFAULT
DEPRECATE
DEPRECATENEXT
DLLINIT
EXIT
EXTRN
FLAG
FLOATINMATH
GONUMBER
GRAPHIC
IGNORE
IMPRECISE
INCAFTER
INCLUDE
INITAUTO
INITBASED
INITCTL
INITSTATIC
INSOURCE
LANGLEVEL
LIBS
LIMITS
LINECOUNT
LINEDIR
LIST
LISTVIEW
MACRO
MAXGEN
MAXMSG
MAXNEST
MAXSTMT
MAXTEMP
MDECK
MS
MSG
MSGSUMMARY
NAMES
NATLANG
NEST
NOT
NUMBER
OBJECT
OFFSET
OVERRIDE
OPTIONS
OR
PP
PPCICS
PPINCLUDE
PLIST
PPMACRO
PPSQL
PPTRACE
PRTYPE
PREFIX
PROBE
PROCEED
PROCESS
QUOTE
Chapter 4. PL/I preprocessors

Include preprocessor

Examples

Include preprocessor options environment variable

Macro preprocessor

Macro preprocessor options

Macro facility options environment variables

SQL preprocessor

Programming and compilation considerations

SQL preprocessor options

SQL preprocessor options environment variable

SQL preprocessor BIND environment variables

Coding SQL statements in PL/I applications

Defining the SQL communications area

Defining SQL descriptor areas

Embedding SQL statements

Using host variables

Determining equivalent SQL and PL/I data types

Determining compatibility of SQL and PL/I data types

Using host structures

Using indicator variables

Host structure example

Using statements and compile-time options

Large Object (LOB) support

Sample programs for LOB support

User defined functions sample programs

Handling SQL error return codes

SQL compatibility and migration considerations

CICS support

Programming and compilation considerations

CICS preprocessor options

CICS preprocessor options environment variables

Coding CICS statements in PL/I applications

Embedding CICS statements

Writing CICS transactions in PL/I

CICS abends used for PL/I programs

CICS run-time user exit

Chapter 5. Compilation output

Using the compiler listing

Compiler output files

Chapter 6. Linking your program

Starting the linker

Statically linking

Linking from the command line

Linking from a make file

Input and output

Search rules

Specifying directories

Filename defaults

Specifying object files

Using response files

Specifying executable output type

Producing an .EXE file

Producing a dynamic link library

Packing executables

Generating a map file

Linker return codes

Chapter 7. Setting linker options

Setting options on the command line

Setting options in the ILINK environment variable

Using the linker

Specifying numeric arguments

Summary of Windows linker options

Windows linker options

/ ?

/ALIGNADDR

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE

/ALIGNFILE
Chapter 2. Compiling your program

This first part of this chapter describes how to compile, link, and run a simple PL/I program. The remainder of the chapter is dedicated to a more detailed description of setting up your compilation environment.

A short practice exercise

Try compiling, linking, and running a simple program to get an idea of how to use PL/I in the Windows environment.

The HELLO program

Here are the steps to make a program that displays the character string “Hello!” on your computer screen.

1. Create the source program
   Create a file, HELLO.PLI, with the following PL/I statements.
   ```pli
   Hello: proc options(main);
     display('Hello!');
   end Hello;
   ```
   Leave the first space of every line blank: by default, the compiler only recognizes characters in columns 2-72. (For additional information, see “MARGINS” on page 82.)
   Save the file to disk.

2. Compile the program
   In a window or full-screen session, go to the directory that contains the HELLO.PLI file and enter the following command:
   ```pli
   pli hello
   ```
   The compiler displays information about the compilation on your screen, and creates the object file (HELLO.OBJ) in the current directory.

3. Link the program
   Without changing directories, enter the following command:
   ```pli
   ilink hello.obj
   ```
   This combines the file HELLO.OBJ with needed library files (as specified by the LIBS compile-time option), producing the file HELLO.EXE (the executable program) in the same directory.
   Since no parameters are specified with the link command, the defaults are used. (The options available with the link command are described in Chapter 6, “Linking your program,” on page 185.)

4. Run the program
   Without changing directories, enter the following command:
   ```pli
   hello
   ```
   This invokes the HELLO.EXE program, which displays `Hello!` on your monitor.
A short practice exercise

To make things easier, programmers often put the commands for compile, link, and run together in a command (CMD) file.

Using compile-time options

As you prepare to compile programs, consider using a subset of the available compile-time options. For a complete description of the compile-time options, including their optional abbreviated forms, see Chapter 3, "Compile-time options," on page 27.

The following example illustrates how to specify options as part of the compilation command:

```plaintext
pli filename (source attributes(full)

source
This option causes your source code and compiler messages to be saved in a compiler listing file (for example, HELLO.LST).

attributes(full)
This option causes a listing of all the attributes in effect for each programmer-defined identifier to be included in the compiler listing.

Using the sample programs provided with the product

Several sample programs have been included with the product, some of which appear in different parts of this book.

For Windows, the sample programs are installed in the ..\SAMPLES directory. A readme file smwread.me. is provided for the sample programs.

Preparing to compile source programs

Before compiling your source program, you should know what structure and format the compiler expects from your source program files.

Program file structure

A PL/I application can consist of several compilation units. You must compile each compilation unit separately and then build the complete application by linking the resulting object files together.

A compilation unit consists of a main source file and any number of include files. You do not compile the include files separately because they actually become part of the main program during compilation. The compiler does not allow DBCS to be used in source file or include file names.

If your program requires %PROCESS or *PROCESS statements, they must be the first lines in your source file. The first line after them that does not consist entirely of blanks or comments must be a PACKAGE or PROCEDURE statement. The last line of your source file that does not consist entirely of blanks or comments must be an END statement matching the PACKAGE or PROCEDURE statement.

The following examples show the correct way to format source files.

Using a PROCEDURE statement with PROCESS:

```plaintext
%PROCESS ;
%PROCESS ;
%PROCESS ;
```
Preparing to compile source programs

/* optional comments */

procedure_Name: proc( ... ) options( ... );
...
end procedure_Name;

Using a PACKAGE statement with PROCESS::

*PROCESS ;
*PROCESS ;
*PROCESS ;

/* optional comments */

package_Name: package exports( ... ) options( ... );
...
end package_Name;

The source file in a compilation can contain several programs separated by
*PROCESS statements. All but the first set of *PROCESS statements are ignored,
and the compiler assumes a PACKAGE EXPORTS(*) statement before the first
procedure.

INCLUDE processing

You can include additional PL/I files at specified points in a compilation unit by
using %INCLUDE statements. For the %INCLUDE statement syntax, see PL/I
Language Reference.

If you specify the file to be included using a string, the compiler searches for the
file exactly as named in that string. If you specify an include file using one of the
more traditional PL/I methods, however, by either using a ddname and member
name or just a member name, the compiler appends a file extension to the member
name.

You can specify which file extensions are appended to the member name by using
the INCLUDE compiler option. For example, if you specify the INCLUDE option
as INCLUDE(EXT(CPY)), when the compiler sees either of the following
statements, it tries to include the file member.cpy.

%include member;
%include ddname(member);

The compiler searches for this file in the following order:
1. The directories specified in the environment variable IBM.DDNAME, if the
   %include statement specified a ddname
2. The directories specified in the environment variable IBM.SYSLIB
3. The directories specified in the environment variable INCLUDE
4. The current directory.

If you specify more than one extension in the INCLUDE compiler option, the
compiler searches all the directories above using the first extension; then does
another pass through all the same directories using the second extension, and so
on.

%LINE directive

The %LINE directive specifies that the next line should be treated in messages and
in information generated for debugging as if it came from the specified line and
file.
Preparing to compile source programs

The characters '%LINE' must be in columns 1 through 5 of the input line for the directive to be recognized (and conversely, any line starting with these five characters is treated as a %LINE directive). The line-number must be an integral value of seven digits or less and the file-specification must not be enclosed in quotes. Any characters specified after the semicolon are ignored.

An example of what these lines should look like can be obtained by compiling a program with the options PPTRACE MACRO and MDECK.

Margins
By default, the compiler ignores any data in the first column of your source program file and sets the right margin 72 spaces from the left.

You can change the default margin setting (see "MARGINS" on page 82). If you choose to keep the default setting, your source code should begin in column 2.

Note: The %PROCESS (or *PROCESS) statement is an exception to the margin rule and must start in the first column. For more information about the %PROCESS statement, see "%PROCESS statement" on page 25.

Program file format
The compiler, running under the Windows operating systems, expects the contents of your source file to consist of ASCII format and CR-LF type. If you created your file on a workstation, the format should be correct; however, if you transfer a file from another machine environment, make sure that the file transfer utility does any needed translation (to ASCII and CR-LF).

The compiler can interpret characters that are in the range '00'X to '1F'X as control codes. If you use characters in this range in your program, the results are unpredictable.

Line continuation
During compilation, any source line that is shorter than the value of the right-hand margin setting as defined by the MARGINS option is padded on the right with blank characters to make the line as long as the right-hand margin setting. For example, if you use the IBM-default MARGINS (2,72), any line less than 72 characters long is padded on the right to make the line 72 characters long.

If long identifier names extend beyond the right margin, you should put the entire name on the next line rather than try to split it between two lines.

If a line of your program exactly reaches the right-hand margin, the last character of that line is concatenated with the first character within the margins of the next line with no blank characters in between.

If you have a string that reaches beyond the right-hand margin setting, you can carry the text of the string on to the following line (or lines). It is recommended that long strings be split into a series of shorter strings (each of which fits on a line) that are concatenated together. For example, instead of coding this:

\begin{verbatim}
do;
  if x > 200 then
    display ('This is a long string and requires more than one line to
\end{verbatim}

1. A CR-LF type file is composed of lines of variable lengths, each delimited by the CR-LF characters. CR and LF are special ASCII characters that signify "Carriage Return" and "Line Feed"—hexadecimal values 0D and 0A, respectively. The compiler interprets CR-LF, LF-CR, CR, or LF as a record delimiter. The hexadecimal value 1A signifies the end of the file.
Preparing to compile source programs

```cpp
type it into my program');
else
display ('This is a short string');
end;

You should use the following sequence of statements:
do;
  if x > 200 then
    display ('This is a long string and requires more than ' || 'one line to type it into my program');
  else
    display ('This is a short string');
end;
```

Setting compile-time environment variables

The way you set compile-time environment variables depends on your operating system.

In Windows, environment variables are set in the System window (to get there, double-click on Main and then on Control Panel). In the System window, click on Set to add a new item to the list of User Environment Variables. Options set in the Windows System window are in effect when you boot your computer unless you override them using a .CMD file or by specifying options on the command line.

For more information on environment variables and how they are used, refer to your system documentation.

The compiler provides several environment variables. They allow you to customize the defaults for:

- Location of compiler input and output
- Compile-time options.

The default location for compiler input and output is the current directory; the IBM-supplied default for each compile-time option is specified in Chapter 3, “Compile-time options,” on page 27.

Some of the compiler environment variables specify a directory path—this should not include a file name or extension. If the path is for compiler input, each individual path (except the last) must be delimited by a semicolon. If the path is for compiler output, only the path to a specific directory is allowed.

**IBM.OPTIONS**

The IBM.OPTIONS environment variable specifies compiler option settings. For example:

```bash
set ibm.options=xref attributes
```

The syntax of the character string you assign to the IBM.OPTIONS environment variable is the same as that required for the compile-time options specified on the PLI command (see “Using the PLI command to invoke the compiler” on page 24).

The defaults together with the changes you apply using this environment variable become the new defaults. Any options you specify on the PLI command or in your source program override these defaults.
Setting compile-time environment variables

**IBM.PPINCLUDE**

The IBM.PPINCLUDE environment variable specifies the include preprocessor option settings. For example:

```
set ibm.ppinclude=id(++include)
```

The syntax of the character string you assign to the IBM.PPINCLUDE environment variable is the same as that required for the compile-time options specified on the PLI command (see "Using the PLI command to invoke the compiler" on page 24).

The defaults together with the changes you apply using this environment variable become the new defaults. Any options you specify on the PP(INCLUDE) option in the IBM.OPTIONS environment variable or the PLI command or in your source program override these defaults.

**IBM.PPMACRO**

The IBM.PPMACRO environment variable specifies the macro facility option settings. For example:

```
set ibm.ppmacro=xref print
```

The syntax of the character string you assign to the IBM.PPMACRO environment variable is the same as that required for the compile-time options specified on the PLI command (see "Using the PLI command to invoke the compiler" on page 24).

The defaults together with the changes you apply using this environment variable become the new defaults. Any options you specify on the PP(MACRO) option in the IBM.OPTIONS environment variable or the PLI command or in your source program override these defaults.

**IBM.PPSQL**

The IBM.PPSQL environment variable specifies the SQL preprocessor option settings. For example:

```
set ibm.ppsql=dbname(employee)
```

The syntax of the character string you assign to the IBM.PPSQL environment variable is the same as that required for the compile-time options specified on the PLI command (see "Using the PLI command to invoke the compiler" on page 24).

The defaults together with the changes you apply using this environment variable become the new defaults. Any options you specify on the PP(SQL) option in the IBM.OPTIONS environment variable or the PLI command or in your source program override these defaults.

**IBM.PPCICS**

The IBM.PPCICS environment variable specifies the CICS preprocessor option settings. For example:

```
set ibm.ppcics=source edf
```

The syntax of the character string you assign to the IBM.PPCICS environment variable is the same as that required for the compile-time options specified on the PLI command (see "Using the PLI command to invoke the compiler" on page 24).
Setting compile-time environment variables

The defaults together with the changes you apply using this environment variable become the new defaults. Any options you specify on the PP(CICS) option in the IBM.OPTIONS environment variable or the PLI command or in your source program override these defaults.

**IBM.SOURCE**

The IBM.SOURCE environment variable specifies the paths for your source program files. For example:

```
set ibm.source=c:\pli\project\updates;\pli\system
```

**IBM.SYSLIB**

The IBM.SYSLIB environment variable specifies the primary input directory search path for include files identified by `%INCLUDE` statements in your source program. For example:

```
set ibm.syslib=c:\pli\project\updates;\pli\system
```

These directories are searched *before* any directories specified in the INCLUDE environment variable.

**IBM.PRINT**

The IBM.PRINT environment variable specifies the path where listing files are written. For example:

```
set ibm.print=c:\pli\project\updates
```

Listing files have the same name as your source program file, with an extension of ASM for the assembler listing and LST for the other listing information.

By default, diagnostic messages and a return code are displayed on your screen.

**IBM.OBJECT**

The IBM.OBJECT environment variable specifies the output directory for object and definition files which have the same name as your source program file, with an extension of OBJ or DEF. For example:

```
set ibm.object=c:\pli\project\updates
```

An object file contains the machine code translation of your PL/I source statements. To make it executable, you must link it with any other OBJ files that comprise your program, and with appropriate library files. For a summary of how to link your program, see Chapter 6, "Linking your program," on page 185.

**IBM.DECK**

The IBM.DECK environment variable specifies the output directory for the modified source file produced by the macro facility. This file is only produced when the MDECK compile-time option is in effect. For example:

```
set ibm.deck=c:\pli\project\updates
```

The output file has the same name as your primary source program file, with an extension of DEK. You can use it as input to a later compilation.
Setting compile-time environment variables

**INCLUDE**

The INCLUDE environment variable specifies the secondary input directory search path for include files identified by `%INCLUDE` statements in your source program. For example:

```
set include=c:\pli\program
```

These directories are searched after any specified in the IBM.SYSLIB environment variable.

**TMP**

The TMP environment variable specifies the input and output directory for any temporary work files that the compiler needs. For example:

```
set tmp=c:\pli\project\updates
```

Do not specify a directory that resides on a Local Area Network (LAN). If you are working with large programs, make sure you set this variable to a location with sufficient free space.

Using the PLI command to invoke the compiler

Use the PLI command to invoke the compiler. You can enter it on the command line or in a CMD file.

```
pli program_file_specification
```

The Windows file specification for your primary source program file. If you omit the extension from your file specification, the compiler assumes an extension of PLI. If you omit the complete path, the current directory is assumed, unless you specify otherwise using IBM.SOURCE.

```
compiler_option
```

One or more compile-time options, described in Chapter 3, “Compile-time options,” on page 27.

The following is an example of the PLI command:

```
pli hello (source
```

You can use a response file to put common options into a file and then use that file to compile various programs. For example, if the file pli.opt contained a list of options, then you could compile the towers sample program as follows:

```
pli towers.pli ( @pli.opt
```

When using response files, remember these guidelines:

- The name of the source file and options can come before the name of the response file, but nothing should follow it.
- A response file can point to another response file.

Where to specify compile-time options

You can specify compile-time options in the three places described in the following sections. Each successive place overrides the options specified in the previous place, starting with the defaults as a base.

*Note:* After PL/I determines the compile-time option settings to use in compilation, individual source statements in your program might further modify the effect of various compile-time options. For example, specifying
OPTION(BYVALUE) in the program takes precedence over the DEFAULT(BYVALUE) compile-time option.

IBM.OPTIONS and IBM.PPxxx environment variables
The first way you can specify options is to set the IBM.OPTIONS environment variable for compile-time options and IBM.PPxxx environment variables for preprocessor options. See “Setting compile-time environment variables” on page 21. Controlling compile-time options with these environment variables overrides the normal option defaults.

PLI command
The second way to specify compile-time options—overriding option defaults and IBM.OPTIONS and IBM.PPxxx—is on the PLI command when you invoke the compiler (see “Using the PLI command to invoke the compiler” on page 24). The options apply only to the current compilation.

%PROCESS statement
The third and final way to specify compile-time options—overriding option defaults, IBM.OPTIONS and IBM.PPxxx and the PLI command—is to use the %PROCESS (or *PROCESS) statement in your PL/I source program. The options apply only to the current compilation.

The following example illustrates the use of the %PROCESS statement:
```pli
%process source margins(1,80);
Hello: proc options(main);
   display('Hello!');
end Hello;
```
You can specify one or more %PROCESS statements, but they must precede all other PL/I source statements, including blank lines.

You must code the percent sign (or the asterisk) of the PROCESS statement in the first column of your source file. The keyword PROCESS can follow in the next column or after any number of blanks. The list of compile-time options on the %PROCESS statement must not extend beyond the default right-hand margin. You can continue the %PROCESS statement onto the next line, but make sure that in doing so you do not split a keyword or value. It is recommended that, instead of wrapping the statement, you code multiple %PROCESS statements, one per line.

Once all %PROCESS statements are interpreted, the rest of the program is read using the margin settings determined after considering the PLI command and the %PROCESS statements. This means that the sample %PROCESS statement shown previously would be processed correctly assuming that the default, MARGINS(2,72), was in effect at compile time.
Where to specify compile-time options
Chapter 3. Compile-time options

This chapter contains detailed compile-time options descriptions, including abbreviations, defaults, and code samples where applicable.

Compile-time option descriptions

There are three types of compiler options; however, 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). The three types of compiler options are:

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 2 lists all the compiler options with their abbreviations (if any) and their IBM-supplied default values. If an option has any suboptions which may 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 pages that follow.

The paragraphs following Table 2 describe the options in alphabetical order. For those options specifying that the compiler is to list information, only a brief description is included; the generated listing is described under “Using the compiler listing” on page 175.

Table 2. Compile-time options, abbreviations, and IBM-supplied defaults

<table>
<thead>
<tr>
<th>Compile-Time Option</th>
<th>Abbreviated Name</th>
<th>Windows Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDEXT</td>
<td>NOADDEXT</td>
<td>-</td>
</tr>
<tr>
<td>AGGREGATE(DECIMAL</td>
<td>HEXADEC)</td>
<td>AG</td>
</tr>
<tr>
<td>ATTRIBUTES[(FULL</td>
<td>SHORT)]</td>
<td>A</td>
</tr>
<tr>
<td>BIFPREC(15</td>
<td>31)</td>
<td>-</td>
</tr>
<tr>
<td>BLANK('c')</td>
<td>-</td>
<td>BLANK('t')</td>
</tr>
<tr>
<td>CASERULES</td>
<td>-</td>
<td>CASERULES(KEYWORD (MIXED))</td>
</tr>
<tr>
<td>CHECK(STORAGE</td>
<td>NOSTORAGE, CONFORMANCE</td>
<td>NOCONFORMANCE)</td>
</tr>
<tr>
<td>CMPAT(LE</td>
<td>V1</td>
<td>V2</td>
</tr>
<tr>
<td>CODEPAGE(n)</td>
<td>CP</td>
<td>CODEPAGE(00819)</td>
</tr>
<tr>
<td>COMPILE</td>
<td>NOCOMPILE[(W</td>
<td>E</td>
</tr>
<tr>
<td>COPYRIGHT('string')</td>
<td>NOCOPYRIGHT</td>
<td>-</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1998, 2012
## Compile-time options

### Table 2. Compile-time options, abbreviations, and IBM-supplied defaults (continued)

<table>
<thead>
<tr>
<th>Compile-Time Option</th>
<th>Abbreviated Name</th>
<th>Windows Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENCY('c')</td>
<td>CURR</td>
<td>CURRENCY($)</td>
</tr>
<tr>
<td>NODBCS</td>
<td>DBCS(JPN</td>
<td>CHS</td>
</tr>
<tr>
<td>DEFAULT(attribute</td>
<td>option)</td>
<td>DFT</td>
</tr>
<tr>
<td>DEPRECA TED(BUILTIN(built-in-name)</td>
<td>ENTRY(entry-name)</td>
<td>INCLUDE(filename)</td>
</tr>
<tr>
<td>DLLINIT</td>
<td>NODLLINIT -</td>
<td>NODLLINIT</td>
</tr>
<tr>
<td>EXIT</td>
<td>NOEXIT -</td>
<td>NOEXIT</td>
</tr>
<tr>
<td>EXTRN(FULL</td>
<td>SHORT) -</td>
<td>EXTRN(FULL)</td>
</tr>
<tr>
<td>FLAG([I</td>
<td>W</td>
<td>E</td>
</tr>
<tr>
<td>FLOATINMATH(ASIS</td>
<td>LONG</td>
<td>EXTENDED) -</td>
</tr>
<tr>
<td>GONUMBER</td>
<td>NOGONUMBER</td>
<td>GN</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>NOGRAPHIC</td>
<td>GR</td>
</tr>
<tr>
<td>IGNORE(ASSERT, DISPLAY, PUT)</td>
<td>NOIGNORE -</td>
<td>NOIGNORE</td>
</tr>
<tr>
<td>IMPRECISE</td>
<td>NOIMPRECISE -</td>
<td>IMPRECISE</td>
</tr>
<tr>
<td>INCAFTER(PROCESS(filename))</td>
<td>-</td>
<td>INCAFTER((PROCESS(&quot;&quot;))</td>
</tr>
<tr>
<td>INCDIR(directory name') -</td>
<td>INCDIR()</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>INCLUDE(filename)</td>
<td>INC(EXT('inc'))</td>
</tr>
<tr>
<td>INITAUTO</td>
<td>NOINITAUTO -</td>
<td>NOINITAUTO</td>
</tr>
<tr>
<td>INITBASED</td>
<td>NOINITBASED -</td>
<td>NOINITBASED</td>
</tr>
<tr>
<td>INITCTL</td>
<td>NOINITCTL -</td>
<td>NOINITCTL</td>
</tr>
<tr>
<td>INITSTATIC</td>
<td>NOINITSTATIC -</td>
<td>NOINITSTATIC</td>
</tr>
<tr>
<td>INSOURCE(FULL</td>
<td>SHORT)</td>
<td>NOINSOURCE</td>
</tr>
<tr>
<td>LANGLVL(NOEXT</td>
<td>OS) -</td>
<td>LANGLVL(OS)</td>
</tr>
<tr>
<td>LIBS -</td>
<td>See &quot;LIBS&quot; on page 73</td>
<td></td>
</tr>
<tr>
<td>LINECOUNT(n)</td>
<td>LC</td>
<td>LINECOUNT(60)</td>
</tr>
<tr>
<td>LINEDIR</td>
<td>NOLINEDIR -</td>
<td>NOLINEDIR</td>
</tr>
<tr>
<td>LIST</td>
<td>NOLIST -</td>
<td>NOLIST</td>
</tr>
<tr>
<td>LISTVIEW(SOURCE</td>
<td>AFTERMACRO</td>
<td>AFTERCICS</td>
</tr>
<tr>
<td>MACRO</td>
<td>NOMACRO</td>
<td>M</td>
</tr>
<tr>
<td>MARGIN('c')</td>
<td>NOMARGIN</td>
<td>MI</td>
</tr>
<tr>
<td>MARGINS(m,n,c)</td>
<td>NOMARGINS</td>
<td>MAR(m,n)</td>
</tr>
<tr>
<td>MAXGEN(n)</td>
<td>MAXGEN(100000) -</td>
<td>MAXGEN(W,250)</td>
</tr>
</tbody>
</table>
### Compile-time options

<table>
<thead>
<tr>
<th>Compile-Time Option</th>
<th>Abbreviated Name</th>
<th>Windows Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXNEST BLOCK(x) DO(y) IF(z)</td>
<td>-</td>
<td>MAXNEST BLOCK(17) DO(17) IF(17)</td>
</tr>
<tr>
<td>MAXSTMT(n)</td>
<td>-</td>
<td>MAXSTMT(4096)</td>
</tr>
<tr>
<td>MAXTEMP(n)</td>
<td>-</td>
<td>MAXTEMP(50000)</td>
</tr>
<tr>
<td>MDECK</td>
<td>NOMDECK</td>
<td>MD</td>
</tr>
<tr>
<td>MSG(390 | \ast)</td>
<td>-</td>
<td>MSG(\ast)</td>
</tr>
<tr>
<td>MSGSUMMARY([XREF | NOXREF])</td>
<td>NOMSGSUMMARY</td>
<td></td>
</tr>
<tr>
<td>NAMES(\text{lower}[ , \text{upper}])</td>
<td>-</td>
<td>NAMES(#@$),#@$)</td>
</tr>
<tr>
<td>NATLANG(ENU | CHS | CHT | DEU | ESP | FRA | JPN | PTB)</td>
<td>-</td>
<td>NATLANG(ENU)</td>
</tr>
<tr>
<td>NEST</td>
<td>NONEST</td>
<td>-</td>
</tr>
<tr>
<td>NOT</td>
<td>-</td>
<td>NOT(\neg)</td>
</tr>
<tr>
<td>NUMBER</td>
<td>NONUMBER</td>
<td>NUM</td>
</tr>
<tr>
<td>OBJECT</td>
<td>NOOBJECT</td>
<td>OBJ</td>
</tr>
<tr>
<td>OFFSET</td>
<td>NOOFFSET</td>
<td>OF</td>
</tr>
<tr>
<td>ONSNAP(\text{STRINGRANGE, STRINGSIZE})</td>
<td>NOONSNAP</td>
<td></td>
</tr>
<tr>
<td>OPTIMIZE(0 | 2)</td>
<td>NOOPTIMIZE</td>
<td>OPT</td>
</tr>
<tr>
<td>OPTIONS([ALL | DOC])</td>
<td>NOOPTIONS</td>
<td>OP</td>
</tr>
<tr>
<td>OR(\text{c})</td>
<td>-</td>
<td>OR(\text{;})</td>
</tr>
<tr>
<td>PP(\text{pp-name})</td>
<td>NOPP</td>
<td>-</td>
</tr>
<tr>
<td>PPCICS(\text{string})</td>
<td>NOPPCICS</td>
<td>-</td>
</tr>
<tr>
<td>PPINCLUDE(\text{string})</td>
<td>NOPPINCLUDE</td>
<td>-</td>
</tr>
<tr>
<td>PPLIST(\text{KEEP} | \text{ERASE})</td>
<td>PPLIST(\text{KEEP})</td>
<td>-</td>
</tr>
<tr>
<td>PPMACRO(\text{string})</td>
<td>NOPPMACRO</td>
<td>-</td>
</tr>
<tr>
<td>PPSQL(\text{string})</td>
<td>NOPPSQL</td>
<td>-</td>
</tr>
<tr>
<td>PTPRACE</td>
<td>NOPTPRACE</td>
<td>-</td>
</tr>
<tr>
<td>PRECTYPE (\text{ANS} | \text{DECIDIG} | \text{DECRESULT})</td>
<td>-</td>
<td>PRECTYPE(\text{ANS})</td>
</tr>
<tr>
<td>PREFIX(\text{condition})</td>
<td>-</td>
<td>See “PREFIX” on page 110</td>
</tr>
<tr>
<td>PROBE</td>
<td>NOPROBE</td>
<td>-</td>
</tr>
<tr>
<td>PROCEED</td>
<td>NOPROCEED([W | E | S])</td>
<td>PRO</td>
</tr>
<tr>
<td>PROCESS([\text{KEEP} | \text{DELETE})]</td>
<td>NOPROCESS</td>
<td>-</td>
</tr>
<tr>
<td>QUOTE(\text{&quot;&quot;})</td>
<td>-</td>
<td>QUOTE(\text{&quot;&quot;})</td>
</tr>
<tr>
<td>REDUCE</td>
<td>NOREDUCE</td>
<td>-</td>
</tr>
<tr>
<td>RESEXP</td>
<td>NORESEXP</td>
<td>-</td>
</tr>
<tr>
<td>RESPECT([\text{DATE})]</td>
<td>-</td>
<td>RESPECT()</td>
</tr>
<tr>
<td>RULES(\text{options})</td>
<td>-</td>
<td>See “RULES” on page 118</td>
</tr>
<tr>
<td>SEMANTIC</td>
<td>NOSEMANTIC([W | E | S])</td>
<td>SEM</td>
</tr>
<tr>
<td>SNAP</td>
<td>NOSNAP</td>
<td>-</td>
</tr>
<tr>
<td>SOSI</td>
<td>NOSOSI</td>
<td>-</td>
</tr>
</tbody>
</table>
## Compile-time options

Table 2. Compile-time options, abbreviations, and IBM-supplied defaults (continued)

<table>
<thead>
<tr>
<th>Compile-Time Option</th>
<th>Abbreviated Name</th>
<th>Windows Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE</td>
<td>NOSOURCE</td>
<td>S</td>
</tr>
<tr>
<td>STATIC(FULL</td>
<td>SHORT)</td>
<td>-</td>
</tr>
<tr>
<td>STMT</td>
<td>NOSTMT</td>
<td>-</td>
</tr>
<tr>
<td>STORAGE</td>
<td>NOSTORAGE</td>
<td>STG</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>NOSYNTAX(W</td>
<td>E</td>
</tr>
<tr>
<td>SYSPARM(string')</td>
<td>-</td>
<td>SYSPARM('')</td>
</tr>
<tr>
<td>SYSTEM(WINDOWS</td>
<td>CICS</td>
<td>IMS</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>NOTERMINAL</td>
<td>TERM</td>
</tr>
<tr>
<td>TEST</td>
<td>NOTEST</td>
<td>-</td>
</tr>
<tr>
<td>USAGE(options)</td>
<td>-</td>
<td>See &quot;USAGE&quot; on page 138</td>
</tr>
<tr>
<td>WIDECHAR(BIGENDIAN</td>
<td>LITTLEENDIAN)</td>
<td>WCHAR</td>
</tr>
<tr>
<td>WINDOW(w)</td>
<td>-</td>
<td>WINDOW(1950)</td>
</tr>
<tr>
<td>XINFO(options)</td>
<td>-</td>
<td>XINFO(NODEF,NOMSG, NOSYMNSYN,NOXML, NOXML)</td>
</tr>
<tr>
<td>XML(CASE(UPPER</td>
<td>ASIS))</td>
<td>-</td>
</tr>
<tr>
<td>XREF[(FULL</td>
<td>SHORT)]</td>
<td>NOXREF</td>
</tr>
</tbody>
</table>

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.

### Rules for using compile-time options

1. If you specify mutually exclusive compile-time options or suboptions, the last one you specify takes effect.

2. If required strings conform to PL/I identifier rules, you do not need to enclose them in quotation marks. The compiler folds these strings to uppercase.

   The following options should have their string specifications enclosed in quotation marks, because the string specifies either special characters or run-time options:
   - CURRENCY
   - DEFAULT(INITFILL)
   - MARGINI
   - NAMES
   - NOT
   - OR

3. If an option has a string enclosed in quotation marks, the string itself cannot contain any quotation marks.

4. If an option has a string enclosed in quotation marks, the string can be specified as a hexadecimal string, for example NOT('aa'x).
ADDEXT

This option specifies whether file extensions (.pli, .cpy) are added by the compiler to source and include file names that have no extensions.

NOADDEXT

File extensions are not added by the compiler. You must specify NOADDEXT when you do a checkout compile on the workstation from within the remote edit/compile environment because no mapping takes place.

ADDEXT

File extensions are added by the compiler. If you specify ADDEXT, the compiler would interpret the command "pli hello" as "pli hello.pli".

ADDEXT and NOADDEXT only affect whether file extensions are added to filenames when the compiler is searching for a file. Compiler output files have file extensions regardless of the setting of this option.
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: NAG, AG

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

DECIMAL
    All offsets in the aggregate listing will be displayed in decimal.

HEXADEC
    All offsets in the aggregate listing will be displayed in hexadecimal.

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 may be inaccurate or specified as *.

A sample listing is shown in "Using the compiler listing" on page 175.
ATTRIBUTES

This option specifies that a table of source-program identifiers and their attributes is included in the compiler listing.

ABBREVIATIONS: NA, A

FULL
List all identifiers and attributes. For an example of the table produced when you select ATTRIBUTES(FULL), see “Using the compiler listing” on page 175.

SHORT
Omit unreferenced identifiers.
BIFPREC

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

\[
\text{BIFPREC}(15)
\]

For best compatibility with PL/I for z/OS compilers, BIFPREC(15) should be used.

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:

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

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

Note that the function `funky` would 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 these three functions: under CMPAT(V1), these array-handling functions return a FIXED BIN(15) result, while under CMPAT(V2) and CMPAT(LE), they return a FIXED BIN(31) result. Under CMPAT(V3), they return a FIXED BIN(63) result.
The BLANK option specifies up to ten alternate symbols for the blank character.

\[ \text{BLANK} \left( -' \ char \ (' \right) \]

**Note:** Do not code any blanks between the quotes.

The IBM-supplied default code point for the BLANK symbol is '09'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*.

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

**DEFAULT:** BLANK('09'x)
Compile-time options

CASERULES

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

| CASERULES | KEYWORD | UPPER | LOWER | START |

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.
CHECK

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

ABBREVIATIONS: STG, NSTG

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

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). 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
The CMPAT option specifies the format used for descriptors generated by the compiler.

ABBREVIATIONS: CMP

LE  Under CMPAT(LE), the compiler generates descriptors in the format defined by the Language Environment product.

V1  Under CMPAT(V1), the compiler generates the same descriptors as would be generated by the OS PL/I Version 1 compiler.

V2  Under CMPAT(V2), the compiler generates the same descriptors as would be generated by the OS PL/I Version 2 compiler when the CMPAT(V2) option was specified.

V3  Under CMPAT(V3), the compiler generates the same descriptors as would be generated by the Enterprise PL/I compiler when the CMPAT(V3) option was specified.

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

The DFT(DESCLIST) option conflicts with the CMPAT(V1), CMPAT(V2), or CMPAT(V3) option, and if it is specified with the CMPAT(V1), CMPAT(V2), or CMPAT(V3) option, a message will be issued and the DFT(DESCLOCATOR) option assumed.
The CODEPAGE option specifies the code page used for:

- conversions between CHARACTER and WIDECHAR
- the default code page used by the PLISAX built-in subroutines

\[\text{CODEPAGE}(ccsid)\]

The supported CCSID's are:

<table>
<thead>
<tr>
<th>CCSID</th>
<th>CCSID</th>
<th>CCSID</th>
<th>CCSID</th>
</tr>
</thead>
<tbody>
<tr>
<td>01047</td>
<td>01145</td>
<td>00273</td>
<td>00297</td>
</tr>
<tr>
<td>01140</td>
<td>01146</td>
<td>00277</td>
<td>00500</td>
</tr>
<tr>
<td>01141</td>
<td>01147</td>
<td>00278</td>
<td>00871</td>
</tr>
<tr>
<td>01142</td>
<td>01148</td>
<td>00280</td>
<td>00819</td>
</tr>
<tr>
<td>01143</td>
<td>01149</td>
<td>00284</td>
<td>00813</td>
</tr>
<tr>
<td>01144</td>
<td>00037</td>
<td>00285</td>
<td>00920</td>
</tr>
</tbody>
</table>

The default CCSID 00819 is the Latin-1 ASCII codepage.
Compile-time options

**COMPILE**

This option specifies that execution of the code generation stage depends on the severity of messages issued prior to this stage of processing.

ABBREVIATIONS: NC, C

**NOCOMPILE**

Compilation halts unconditionally after semantic checking.

**NOCOMPILE(S)**

Compilation halts if a severe or unrecoverable error is detected.

**NOCOMPILE(E)**

Compilation halts if an error, severe error, or unrecoverable error is detected.

**NOCOMPILE(W)**

Compilation halts if a warning, error, severe error, or unrecoverable error is detected.

**COMPILE**

Equivalent to NOCOMPILE(S).

If the compilation is terminated by the NOCOMPILE option, whether or not listings are produced depends on when the compilation stopped. For example, cross-reference and attribute listings should be produced with the NOCOMPILE option, but an error might occur during semantic checking that stops those listings from being produced.
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.

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

CURRENCY

This option allows you to specify a unique character for the dollar sign.

\texttt{CURRENCY\{x\}}

\(x\) Character that you want the compiler and runtime to recognize and accept as the dollar sign in picture strings.

DEFAULT: \texttt{CURRENCY\{\$\}}
The DBCS option determines what, if any, bytes are accepted as forming DBCS.

When the DBCS option is specified, if the GRAPHIC option is also specified, then appropriate DBCS characters from the indicated language will be accepted in identifiers, literals etc.
Compile-time options

**DEFAULT**

This 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.
**Compile-time options**

ABBREVIATIONS: DFT, ASGN, NONASGN, CONN, NONCONN

**ALIGNED or UNALIGNED**
This suboption allows you to force 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.

**ANS or IBM**
Use IBM or ANS SYSTEM defaults. The arithmetic defaults for IBM and ANS are the following:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>DEFAULT(IBM)</th>
<th>DEFAULT(ANS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIXED DECIMAL</td>
<td>(5,0)</td>
<td>(10,0)</td>
</tr>
<tr>
<td>FIXED BINARY</td>
<td>(15,0)</td>
<td>(31,0)</td>
</tr>
<tr>
<td>FLOAT DECIMAL</td>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>FLOAT BINARY</td>
<td>(21)</td>
<td>(21)</td>
</tr>
</tbody>
</table>

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

**ASCII or EBCDIC**

Use this option to set the default for the character set used for the internal representation of character problem program data.

Specify EBCDIC only when compiling programs that depend on the EBCDIC 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 '313133'X
'123'E is the same as 'F1F1F3'X

**ASSIGNABLE or NONASSIGNABLE**
This option applies only to static variables. The compiler flags statements in which NONASSIGNABLE variables are the targets of assignments. If you are porting code to the mainframe, this option flags statements that would otherwise raise a protection exception (if your program is reentrant).

**BIN1ARG or NOBIN1ARG**
This suboption controls how the compiler handles 1-byte REAL FIXED BIN arguments passed to an unprototyped function.
Under BIN1ARG, the compiler will pass a FIXED BIN argument as is to an unprototyped function.

But under NOBIN1ARG, the compiler will assign 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 specified DEFAULT(BIN1ARG), the compiler would pass 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 specified DEFAULT(NOBIN1ARG), the compiler would pass the address of a 1-byte FIXED BIN(15) argument to both routines.

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

BIN1ARG is the default.

**BYADDR or BYVALUE**

Set the default for whether arguments or parameters are passed by address or by value. BYVALUE applies only to certain arguments and parameters. See the *PL/I Language Reference* for more information.

**CONNECTED or NONCONNECTED**

Set 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.

**DESLIST or DESCLOCATOR**

When you specify DEFAULT(DESLIST), the compiler generates code in the same way as previous workstation product releases (all descriptors are passed in a list as a ‘hidden’ last parameter).

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

DESLIST is the default.

**DESCRIPTOR or 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 yields errors under the following conditions:

- For PROCEDURE statements, NODESCRIPTOR is invalid if any of the parameters have:
  - An asterisk (*) specified for the bound of an array, the length of a string, or the size of an area
  - The NONCONNECTED attribute
  - The UNALIGNED BIT attribute
Compile-time options

- 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.

**DUMMY(ALIGNED or 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:

```pli
  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 specified DEFAULT(DUMMY(ALIGNED)), a dummy argument would be created, while if you specified DEFAULT(DUMMY(UNALIGNED)), no dummy argument would be created.

DUMMY(ALIGNED) is the default.

**E(IEEE or HEXADEC)**

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.

DFT( E(HEXADEC) ) is useful in developing and testing 390 applications on the workstation. The statement "put skip edit(x) ( e(15,8));" would produce no messages on 390, but, by default, it would be flagged under Intel and AIX. Specifying DFT(E(HEXADEC)) would fix this.

IEEE is the default.

**EVENDEC or 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 (just as it would not be raised if you were using mainframe PL/I).

EVENDEC is the default.
IEEE or HEXADEC
IEEE specifies that floating-point data is held in storage using native IEEE format. HEXADEC indicates that storage of floating-point data is identical to the mainframe environment.

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

If you specify INITFILL with a hexadecimal value (nn), that value is replicated and fills storage for all automatic variables. If you do not enter a hexadecimal value, the default is '00'x. NOINITFILL does no initialization of these variables. INITFILL can cause programs to run significantly slower and should not be specified in production programs. During program development, however, it is useful for detecting uninitialized automatic variables.

NOINITFILL is the default.

INLINE or 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 on how inlining can improve the performance of your application, see Chapter 17, “Improving performance,” on page 355.

LINKAGE
The linkage convention for procedure invocations is:

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

SYSTEM
All parameters are passed on the stack, but the calling function cleans up the stack.

STDCALL
The standard linking convention for Windows APIs. This linkage convention is used under Windows and passes all parameters on the stack. The called function cleans up the stack.

CDECL
All parameters are passed on the stack, but the calling function cleans up the stack. External names have _ applied as a prefix.

OPTIONS(COBOL) implies LINKAGE(SYSTEM) unless a linkage is specified on the entry DCL or PROC statement.

For more detailed information on linkage conventions, see Chapter 21, “Calling conventions,” on page 395.

LOWERINC or UPPERINC
If you specify LOWERINC, the compiler accepts lowercase filenames for INCLUDE files. If you specify UPPERINC, the compiler accepts uppercase filenames for INCLUDE files.

LOWERINC is the default.

NATIVE or NONNATIVE
This option affects only the internal representation of fixed binary, ordinal,
Compile-time options

offset, area, and varying string data. When the NONATIVE 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 non-native 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:

- Both the NATIVE and NATIVEADDR suboptions
- Both the NONNATIVE and NONNATIVEADDR suboptions.

Other combinations produce unpredictable results.

NATIVEADDR or NONNATIVEADDR

This option 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:

- Both the NATIVE and NATIVEADDR suboptions
- Both the NONNATIVE and NONNATIVEADDR suboptions.

Other combinations produce unpredictable results.

NULLSTRADDR or NONULLSTRADDR

This suboption controls how the compiler handles null strings when 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 will pass 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(NULL|SYSNULL)

This suboption controls how the compiler handles null strings when 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

NULLSTRPTR(NULL) is the default.

NULLSYS or 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 mainframe PL/I, specify NULL370.

NULLSYS is the default.
Compile-time options

ORDER or REORDER
Affects optimization of the object code. Specifying REORDER allows more optimization of your object code, see Chapter 17, “Improving performance,” on page 355.

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

OVERLAP or 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. Chapter 17, “Improving performance,” on page 355.

PSEUDODUMMY or NOPSEUDODUMMY
This suboption determines if 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 or 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 or NORETCODE
If you specify RETCODE, any external procedure that does not have the RETURNS attribute returns an integer value obtained by invoking the PLIRETV built-in function just prior to returning from that procedure. This makes such procedures behave like similar procedures invoked from COBOL on the mainframe.

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

RETURNS (BYVALUE or BYADDR)
Sets the default for how values are returned by functions. See the PL/I Language Reference for more information.

RETURNS(BYVALUE) is the default. You should 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.

SHORT(HEXADEC or IEEE)
This suboption improves compatibility with other unix PL/I compilers. SHORT (HEXADEC) indicates that FLOAT BIN (p) is to be mapped to a short (4-byte) floating point number for p ≤ 21. SHORT (IEEE) indicates that FLOAT BIN (p) is to be mapped to a short (4-byte) floating point number for p ≤ 24. SHORT (HEXADEC) is the default.
Compile-time options

DEFAULT (IBM ASCII ASSIGNABLE BYADDR NONCONNECTED DESCRIPTOR
NATIVE NATIVEADDR NOINLINE REORDER LINKAGE(OPTLINK) IEEE
EVENDEC BIN1ARG NOINITFILL ORDINAL(MIN) NOOVERLAP
NULLSTRADDR NULLSTRPTR(NULL) NULLSYS NONRECURSIVE DESCLIST
RETURNS(BYVALUE) SHORT(HEXADEC) DUMMY(ALIGNED) LOWERINC
NORETCODE ALIGNED E(IEEE) PSEUDODUMMY).
DEPRECATE

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

```
DEPRECATE ( \[ built-in-name \] )

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 attribute.

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 specifications are invalid:

- DEPRECATE
Compile-time options

- DEPRECATE(BUILTIN)

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

  Specifying a suboption a second time replaces the previous specifications.

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

  Examples:
  - The following specifications are equivalent:

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

  - In the following example, x is replaced by y:

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

  The default is DEPRECATE(BUILTIN()  ENTRY()  INCLUDE()  STMT()  VARIABLE()).
The purpose of specifying the DEPRECATENEXT option and usage of this option is the same with 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 53.
Compile-time options

**DLLINIT**

This option is used to identify whether the resulting object files are to be used in executable (.EXE) or dynamic link library files (.DLL).

NODLLINIT

This option must be in effect for all compilations used to build an .EXE file.

DLLINIT

This option must be specified in at least one of your compilations when you use the object files produced by those compilations to build a .DLL.
The EXIT option enables the compiler user exit to be invoked.

```
NOEXIT
EXIT
(____inparm_string____)
```

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

For more information, see Chapter 18, “Using user exits,” on page 369.
Compile-time options

**EXTRN**

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

```
EXTRN ( FULL )
```

**FULL**

EXTRNs are emitted for all declared external entry constants.

**SHORT**

EXTRNs are emitted only for those constants that are referenced. This is the default.
The FLAG option specifies the minimum severity of error that requires a message to be listed in the compiler listing.

ABBREVIATIONS:

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.
Compile-time options

FLOATINMATH

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

\[ \text{FLOATINMATH} \left( \begin{array}{c}
\text{ASIS} \\
\text{LONG} \\
\text{EXTENDED}
\end{array} \right) \]

\textbf{ASIS}

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

\textbf{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.

\textbf{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 \( p \leq 6 \), long floating-point precision if \( 6 < p \leq 16 \) and extended floating-point precision if \( p > 16 \).

A FLOAT BIN expression with precision \( p \) has short floating-point precision if \( p \leq 21 \), long floating-point precision if \( 21 < p \leq 53 \) and extended floating-point precision if \( p > 53 \).
Compile-time options

GONUMBER

This option creates a statement number table as part of the object file. This table is useful for debugging purposes.

ABBREVIATIONS: NGN, GN
Compile-time options

GRAPHIC

This option specifies that double-byte characters in the source program are present.

ABBREVIATIONS: NGR, GR

You must specify GRAPHIC if you use any of the following in your source program:
- DBCS identifiers
- DBCS in comments
- Graphic string constants
- Mixed string constants
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.

**IGNORE**

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.
Compile-time options

IMPRECISE

This option determines the precision of floating-point results and the location at which floating-point interrupts are reported.

ABBREVIATIONS: IMP, NIMP

IMPRECISE

Precision of floating-point results might not be IEEE conforming and the location of floating-point interrupts might not be precise. The loss of precision is negligible for most applications. The location of interrupt might be close to the interruption point or might be far from the interruption point, perhaps in another block.

Use of this option produces smaller object code that runs faster. It is recommended for your production programs.

NOIMPRECISE

Precision of floating-point results is IEEE conforming and the precise location of floating-point interrupts is required. This option produces code that runs slower and is recommended only, if at all, during program development.

Although NOIMPRECISE does provide better floating-point error detection than IMPRECISE, the Windows operating system does not allow immediate detection of floating-point exceptions. If you have a statement in your program that is likely to raise a floating-point exception, you can avoid this detection problem by enclosing the statement, by itself, in a BEGIN block.
This option allows you to specify a file to be included after a particular statement in your source program.

\[\text{INCAFTER}\{\text{PROCESS}\{\text{filename}\}\}\]\n
filename

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

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

Consider the following example:

\[\text{INCAFTER(PROCESS(DFTS))}\]

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

This option specifies the file name extensions under which include files are searched. You specify the file name on the %INCLUDE statement and the directory search path on the IBM.SYSLIB or INCLUDE environment variables.

```
#include ( ext(Cop Inc '2++' Mac) )
```

ABBREVIATIONS: INC

The extension string (see the note on strings in step 2 on page 30 under “Rules for using compile-time options”) can be up to 31 characters long, but it is truncated to the first three characters.

If you specify more than one file name extension, the compiler searches for include files with the left most extension you specify first. It then searches for extensions which you specified from left to right. You can specify a maximum of 7 extensions.

DEFAULT: INCLUDE(EXT('INC' 'CPY' 'MAC')).

Do not use 'PLI' as an extension for include files.

Examples:

In this first example, the compiler searches for include files with file name extensions of COP, INC, 2++, and MAC in that order.

```
include ( ext(Cop Inc '2++' Mac) )
```

In the following example, the compiler searches for include files without file name extensions first, and then for those with file name extensions of INC, CPY, and MAC.

```
include (ext(' ',Inc,Cpy,Mac))
```
Under INITAUTO, the compiler adds an INITIAL attribute to an AUTOMATIC variable that does not have an INITIAL attribute.

The compiler determines the INITIAL values according to the data attributes of the variable:
- 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

NOINITAUTO is the default.

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.
Compile-time options

INITBASED

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

NOINITBASED is the default.

INITBASED 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.
INITCTL

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

NOINITCTL is the default.

INITCTL 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.
Compile-time options

INITSTATIC

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

NOINITSTATIC is the default.

The INITSTATIC option could cause some objects larger and some compilations to consume more time, but should otherwise have no impact on performance.
The INSOURCE option specifies that the compiler should include a listing of the source program before the macro preprocessor translates it.

ABBREVIATIONS: NIS, IS

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.

The INSOURCE listing has no effect unless the MACRO option is in effect.

If the INSOURCE option is in effect, and if the compiler invokes more than one of the macro, CICS or SQL preprocessors, the listing contains the source for each of these preprocessors.

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

```plaintext
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 would 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).
Compile-time options

**LANGLVL**

This option specifies the level of the PL/I language definition that you want the compiler to accept. The compiler flags any violations of the specified language definition.

```plaintext
LANGLVL
```

**NOEXT**

No extensions beyond the language level specified are allowed.

**OS**

The ENVIRONMENT options unique to the Windows environment are allowed.
LIBS

This option specifies whether or not the compiler should generate information in
the object file that names the default libraries that are to be searched at link time in
order to resolve references to external entries and data.

LIBS

Same as specifying LIBS(SINGLE DYNAMIC)

LIBS(SINGLE DYNAMIC)

Specifies that default libraries searched at link time are the single-threading
PL/1 libraries:
- On Windows, these are ibmws20i.lib, ibmwstbi.lib, hepws20i.lib, and
  kernel32.lib.

LIBS(MULTI DYNAMIC)

Specifies that default libraries searched at link time are the multi-threaded
PL/1 libraries:
- On Windows, these are ibwm20i.lib, ibmwmtbi.lib, hepwm20i.lib, and
  kernel32.lib.

LIBS(SINGLE STATIC)

Specifies that default libraries searched at link time are the static,
non-multithreading libraries:
- On Windows, these are ibmws20.lib, ibmws35.lib ibmwstb.lib,
  hepws20.lib, and kernel32.lib.

LIBS(MULTI STATIC)

Specifies that default libraries searched at link time are the static,
multi-threaded libraries. This means the library will be statically linked into
the user module.
- On Windows, these are ibwm20.lib, ibwm25.lib ibmwmtb.lib,
  hepwm20.lib, and kernel32.lib.

You should specify the SINGLE suboption only if your application uses no
multithreading language and specify the MULTI suboption when your application
contains any PL/I multithreading language.

You can specify LIBS(MULTI) when no multithreading language is used, however,
this causes your application to run more slowly than it would with LIBS(SINGLE).
LIMITS

This option specifies various implementation limits.

EXTNAME
Specifies the maximum length for 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. If FIXEDDEC(15,31) is specified, then you may declare FIXED DECIMAL variables with precision greater than 15, but unless an expression contains an operand with precision greater than 15, all arithmetic will done using 15 as the maximum precision.

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.

The default is FIXEDDEC(15,15).

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

If FIXEDBIN(31,63) is specified, then you may declare 8-byte integers, but unless an expression contains an 8-byte integer, all arithmetic will done using 4-byte integers.

FIXEDBIN(63,31) is not allowed.

The default is FIXEDBIN(31,31).

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 7.

DEFAULT: LIMITS(EXTNAME(100) FIXEDBIN(31,31) FIXEDDEC(15,15) NAME(100))
LINECOUNT

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

ABBREVIATIONS: LC

The value of \( n \) can be from 1 to 32,767.

DEFAULT: LINECOUNT(60)
Compile-time options

**LINEDIR**

This option specifies that the compiler should accept %LINE directives.

If the LINEDIR option is specified, the compiler will reject all %INCLUDE statements.

DEFAULT: NOLINEDIR
LIST

This option causes an object module listing to be produced. This listing is in a form similar to assembler language instructions.

The object listing is produced in a separate file with an extension of .asm.

Assembler listings do not always compile. A sample listing is shown in "Using the compiler listing" on page 175.
Compile-time options

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.

SOURCE
Causes the source listing to show the unadulterated source, and it causes IBM
Debugger 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 it causes IBM
Debugger to bring up this as the source view if the SEPARATE suboption of
the TEST compiler option is also specified.

AALL may 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 it causes IBM
Debugger to bring up this as the source view if the SEPARATE suboption of
the TEST compiler option is also specified.

ACICS may 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 it causes IBM
Debugger to bring up this as the source view if the SEPERATE suboption of
the TEST compiler option is also specified.

AMACRO may 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 it causes IBM
Debugger to bring up this as the source view if the SEPERATE suboption of
the TEST compiler option is also specified.

ASQL may be used as an abbreviation for AFTERSQL

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

As an example of the differing effects of the AFTERMACRO, AFTERSQL and
AFTERALL suboptions suppose the PP option was PP( MACRO('INCONLY'), SQL,
MACRO). Then:

* Under LISTVIEW(AFTERMACRO), the "source" in the listing and in the IBM
  Debugger 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 IBM Debugger 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 since the MACRO preprocessor is the last in the PP option
The MACRO option causes the macro facility to be invoked prior to compilation. If both MACRO and PP(MACRO) are specified, the macro facility is invoked twice. When the MACRO option is used, MACRO('macro-options') is inserted into the PP option.

ABBREVIATIONS: NM, M

For example, if the following compile-time options are specified:
MDECK NOINSOURCE MACRO PP(MACRO SQL)

The PP option is modified and effectively becomes:
PP (MACRO MACRO SQL)

See also “PP” on page 102.
This option specifies the margin indicator used in the source listing produced.

ABBREVIATIONS: MI('char')

The character, char, is inserted in the positions immediately to the left and right of both side margins, making any source code outside of the margins easily detected.

DEFAULT: MARGINI(' ')

Using the default specifies that left and right source margins are shown in the listing by blank columns.

For a sample listing, see "Using the compiler listing" on page 175.
MARGINS

This option sets the margins within which the compiler interprets the source code in your program file. Data outside these margins is not interpreted as source code, though it is included in your source listing if you request one.

\[ \text{MARGINS}(m,n,c) \]

ABBREVIATIONS: 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.

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

- **c**: The column number of the ANSI printer control character. It must not exceed 200 and should be outside the values specified for \( m \) and \( n \). A value of 0 for \( c \) indicates that no ANSI 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 PL/I Language Reference).

DEFAULT: MARGINS (2 72)
The 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 number of intermediate language statements generated for any user statement may vary depending on the compiler release, the compiler maintenance level, and the compiler options in effect. This option is intended to be used only to help find statements for which excessive amounts of code are generated and which thus may indicate that they are perhaps are poorly coded.

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

The default is MAXGEN(100000).
**Compile-time options**

**MAXMSG**

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

- Count all messages.
- Count all except information messages.
- Count all except warning and information messages.
- Count only severe error and unrecoverable error messages.
- 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. The value of \( n \) can range from 0 to 32767. If you specify 0, the compilation terminates when the first error of the specified severity is encountered.

DEFAULT: MAXMSG( W 250 )
MAXNEST

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

```
MAXNEST ( BLOCK (17) DO (17) IF (17) )
```

**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 of any nesting limit must be between 1 and 50 (inclusive).

The default is MAXNEST( BLOCK(17) DO(17) IF(17) ).
Compile-time options

MAXSTMT

Under the MAXSTMT option, if the MSG(390) option is also in effect, the compiler will flag any block that has more than the specified number of statements. On Windows, however, optimization of such a block will not be turned off.

\[ \text{MAXSTMT}(\text{size}) \]

DEFAULT: MAXSTMT( 4096 )
The MAXTEMP option determines when the compiler flags statements using an excessive amount of storage for compiler-generated temporaries.

```
MAXTEMP <max>
```

`max` is 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 those 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 may be able to reduce the amount of stack storage required by your code.
Compile-time options

MDECK

MDECK

This option specifies that the macro facility output source is written with the file extension of .DEK and the file is put in the current directory.

ABBREVIATIONS: NMD, MD

MDECK is ignored if NOMACRO is in effect. See “MACRO” on page 80 for an example.
MSG

This option controls when the compiler will issue messages for conversions that will be done via a library call.

* Causes the compiler to issue warning messages for conversions that will be done via a library call if and only if they would be done via a library call on the platform where the code is compiled.

390 Causes the compiler to issue warning messages for conversions that will be done via a library call if and only if they would be done via a library call on 390.
MSGSUMMARY

The MSGSUMMARY option determines whether the compiler adds a summary of all messages that are issued during the 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 with the one added when MSGSUMMARY(XREF) 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.
Names

This 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.

```
NAMES (extraling_char, upp_extraling_char)
```

**extraling_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 characters 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.

DEFAULT: NAMES("#@$" "#@$")

Examples:

```
names ("äöüß" "ÄÖÜß")
```
**NATLANG**

This option sets the national language to be used for compiler messages and listings.

CHS
Chinese simplified

CHT
Chinese traditional

DEU
German

ENU
US English, mixed case.

ESP
Spanish

FRA
French

JPN
Japanese

PTB
Brazilian Portuguese

DEFAULT: NATLANG(ENU)
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.

For an example of the source listing, see “Using the compiler listing” on page 175.
Compile-time options

NOT

This option specifies up to seven symbols, any one of which is interpreted as the logical NOT sign.

not('char')

char

A single character symbol. You must not specify any of the 26 alphabetic, 10 digit, and special characters defined in the PL/I Language Reference, except for the logical NOT sign (^).

DEFAULT: NOT (^)

The PL/I default code point for the NOT symbol has the hexadecimal value 5E, which on many terminals will appear as the logical NOT symbol (^).

If you are invoking the compiler from the commandline and specifying a caret (^) as part of the NOT option, you must precede the caret with another caret.

Examples:

not('\')
not('^')

If you are invoking the compiler and specifying any compile-time options that use vertical bars (|) or a caret (^) on the command line, use double quotes around the character.
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, 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 the compilation.

Note that if a preprocessor has been used, more than one line in the source listing may 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 would all be identified by one line and file number.

NUMBER and STMT are mutually exclusive. Specifying NONUMBER implies STMT.

ABBREVIATIONS: NUM, NNUM
Compile-time options

OBJECT

This option specifies whether object code is produced.

ABBREVIATIONS: OBJ, NOBJ

The module is saved in the current directory.
OFFSET

This option specifies whether or not the compiler produces an assembler-like
listing file with the extension .cod.

ABBREVIATIONS: OF, NOF

The .cod file contains the offset and machine code for every instruction generated.
Use the sample program cod2off to reduce the size of this file to a listing of the
offset for the start of each statement in every block of the compilation.
For a PROCEDURE with the OPTIONS(MAIN) attribute, the ONSNAP option specifies that the compiler should insert an ON STRINGRANGE SNAP; or 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

This option specifies the type of optimization required.

ABBREVIATIONS: NOPT, OPT

NOOPTIMIZE or OPTIMIZE(0)
Use either of these options to produce standard optimization of the object code, allowing compilation to proceed as quickly as possible.

OPTIMIZE(TIME) or OPTIMIZE(2)
Use either of these options to cause extended optimizations of the object code and produce faster running object code.

Optimization requires additional compile time, but usually results in reduced run time.

Inlining occurs only under optimization.

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 compile.

Compile-time options

OPTIONS

This option produces a listing of all compile-time options in effect for the compilation (see Chapter 5, “Compilation output,” on page 175 for an example).

ABBREVIATIONS: NOP, OP

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.
This option specifies up to seven symbols, any one of which is interpreted as the logical OR sign (|). These symbols are also used as the concatenation symbol (when paired).

\[ \text{OR}(\text{char}) \]

A single character symbol. You must not specify any of the 26 alphabetic, 10 digit, and special characters defined in the PL/I Language Reference, except for the logical OR sign (|).

If you are invoking the compiler and specifying a vertical bar (|) on the command line as part of the OR option, you must precede the vertical bar with a caret (^).

DEFAULT: OR ('|')

The PL/I default code point for the OR symbol (|) is hexadecimal 7C.

Examples:
\begin{verbatim}
  or('\\')
  or('|')
\end{verbatim}

If you are invoking the compiler and specifying any compile-time options that use vertical bars (|) or a caret (^) on the command line, use double quotes around the character.
Compile-time options

**PP**

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

```
pp-name (pp-string)
```

- **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 quotes, 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)`.

**DEFAULT:** **NOPP**

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.

The same preprocessor can be specified multiple times, and you can specify a maximum of 31 preprocessor steps.

The MACRO option and the **PP(MACRO)** option both cause the macro facility to be invoked prior to compilation. If both MACRO and **PP(MACRO)** are specified, the macro facility is invoked twice.

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 specified **PP(MACRO SQL('OPTIONS'))** and **PP(MACRO SQL('OPTIONS DATE(ISO)'))**, then the resulting PP option would be **PP( MACRO SQL('OPTIONS') MACRO SQL('OPTIONS DATE(ISO)'))** and both the MACRO and SQL preprocessor would be invoked twice. If you were doing this in an attempt 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 via the **PPSQL** option, i.e. specify **PP(MACRO SQL) PPSQL('OPTIONS DATE(ISO)')**.

You can specify a maximum of 31 preprocessors.

**Examples:**

The following example invokes the PL/I macro facility, the SQL preprocessor, and then the PL/I macro facility a second time.

```
pp(macro('x') sql('dbname(sample)') macro)
```
The PPCICS option specifies options to be passed to the CICS preprocessor if it is invoked.

So, 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 could specify this option in the installation options exit. Then whenever you specified PP(CICS), the set of options specified in the PPCICS option would 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 PPINCLUDE option specifies options to be passed to the INCLUDE preprocessor if it is invoked.

\[
\text{NOPPINCLUDE} \quad \text{PPINCLUDE} \quad (-'options \ string'-) \\
\]

So, 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 could specify this option in the installation options exit. Then whenever you specified PP(INCLUDE), the set of options specified in the PPINCLUDE option would 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 PPLIST option controls whether the compiler keeps or erases the part of the listing 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.
The PPMACRO option specifies options to be passed to the MACRO preprocessor if it is invoked.

So, 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 could specify this option in the installation options exit. Then whenever you specified the MACRO or PP(MACRO) options, the set of options specified in the PPMACRO option would 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 PPSQL option specifies options to be passed to the SQL preprocessor if it is invoked.

PPSQL(—'options string'—)

So, specifying PPSQL('ONEPASS') PP(SQL) is the same as specifying PP(SQL('ONEPASS')).

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

Also, any options specified when the preprocessor is invoked overrule those specified in the PPSQL option. So, specifying PPSQL('ONEPASS') PP(SQL('TWOPASS')) is the same as specifying PP(SQL('ONEPASS TWOPASS')) or the even simpler PP(SQL('TWOPASS')).
Compile-time options

**PPTRACE**

This option specifies that when a DECK file is written for a preprocessor, every non-blank 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 non-blank line should be attributed.

PPTRACE should be used only with preprocessors other than those that are integrated with the PL/I compiler.
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 \( \text{BIF}(x, y, p) \) and in \( \text{BIF}(x, y, p, 0) \) 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 \( \text{BIF}(x, y, p, q) \) if \( q \) is not-zero, then 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 \( t = 1 + \text{ceil}(p / 3.32) \) and \( u = \text{ceil}(q / 3.32) \). In this case, \( x, y, p \) and \( q \) are effectively all converted to decimal (in contrast to the next suboption which converts only \( x \) and \( y \) to decimal and does so even if \( q \) is zero). The compiler will issue the Informational message IBM1053 in this situation.

**DEC_DIGIT**
Under PRECTYPE(DEC_DIGIT), the value \( p \) in \( \text{BIF}(x, y, p) \) and in \( \text{BIF}(x, y, p, 0) \) 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 \( s = \text{ceil}(3.32^*p) \)).

For an instance of \( \text{BIF}(x, y, p, q) \) where \( q \) is not-zero, the results under PRECTYPE(DEC_DIGIT) are the same as described below under PRECTYPE(DECRESULT).

**DECRESULT**
Under PRECTYPE(DECRESULT), the value \( p \) in \( \text{BIF}(x, y, p) \) and the values \( p \) and \( q \) in \( \text{BIF}(x, y, p, q) \) 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 \( y \).

PRECTYPE(ANS) is the default.
**PREFIX**

This 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/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 compile.

**DEFAULT:** PREFIX(CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE UNDERFLOW ZERODIVIDE)

Examples:

Given the following source:

```plaintext
(stringsize):
  name: proc options (reentrant reorder);
end;
```

The option prefix (size nounderflow) logically changes the program to the following:

```plaintext
(size nounderflow):
  (stringsize):
  name: proc options (reentrant reorder);
end;
```
PROBE

This option controls the generation of stack probes which are extra instructions generated by the compiler whenever the stack can be extended by more than 2K bytes. This extra code causes a protection exception if there is not enough storage available on the stack.

PROBE

Specifies that stack probes are generated.

NOPROBE

No generation of stack probes.

A program that requires considerable automatic storage, but is linked with an insufficient stack size, produces exceptions and might go into an infinite loop unless stack probes are generated. The presence of stack probes decreases performance in non-multithreading programs that are properly linked.
Compile-time options

**PROCEED**

This option determines whether or not processing (by a preprocessor or the compiler) continues depending on the severity of messages issued by previous preprocessors.

ABBREVIATIONS: PRO, NPRO

**PROCEED**

The invocation of preprocessors and the compiler continue despite any messages issued by preprocessors prior to this stage.

**Noproceed (S)**

The invocation of preprocessors and the compiler does not continue if a severe or unrecoverable error is detected in this stage of preprocessing.

**Noproceed (E)**

The invocation of preprocessors and the compiler does not continue if an error, severe error, or unrecoverable error is detected in this stage of preprocessing.

**Noproceed (W)**

The invocation of preprocessors and the compiler does not continue if a warning, error, severe error, or unrecoverable error is detected in this stage of preprocessing.
**PROCESS**

The PROCESS option determines if *PROCESS statements are allowed and, if they are allowed, if 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.
The QUOTE option specifies an alternate symbol that can be used as the quote character.

Note: Do not code any blanks between the quotes.

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.
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 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.

When you specify the NOREDUCE option, BY NAME assignments that can be reduced to aggregate moves are not reduced if the elements that would be moved together had the AREA or VARYING, or VARYINGZ 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 is run 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 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 causes four assignments to be generated, and the padding byte is unchanged.

However, under REDUCE, the assignment is reduced to one operation, but the padding byte is zeroed out.
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 would not flag the following statement (and the ZERODIVIDE exception would be raised at run time)

```pli
if preconditions_not_met then
    x = 1 / 0;
```
RESPECT

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 causes the compiler to ignore any specification of the DATE attribute and the DATE attribute, therefore, would not be applied to the result of the DATE built-in function.
RULES

This option allows or disallows certain language capabilities and allows you to choose semantics when alternatives are available. It can help you diagnose common programming errors.
Compile-time options

**IBM or 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 *pre-Enterprise PL/I Language Reference*.
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *pre-Enterprise 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 declares.
- If the result of any precision-handling built-in function (ADD, BINARY, etc.) 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, the result is always SIGNED.
- Even when you add, multiply, or divide two UNSIGNED FIXED BIN operands, the result has the SIGNED attribute.
- Even when you apply the MOD or REM built-in functions to two UNSIGNED FIXED BIN operands, the result has the SIGNED 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, etc.) 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, then the result is also 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.

**BYNAME or NOBYNAME**
Specifying NOBYNAME causes the compiler to flag all BYNAME assignments with an E-level message.

**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.

**DECSIZE or 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) may cause the compiler to produce a large number of messages since if SIZE is disabled, then any statement of the form \( X = X + 1 \) will be flagged if \( X \) is FIXED DECIMAL.

**ELSEIF or 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.

**EVENDEC or NOEVENDEC**

Specifying NOEVENDEC causes the compiler to flag any FIXED DECIMAL declaration that specifies an even precision.

**GOTO|NOGOTO**

Specifying NOGOTO causes all GOTO statements to be flagged except for those out of BEGIN blocks.

**LAXBIF or 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.

**GLOBALDO|NOGLOBALDO**

Specifying NOGLOBALDO causes the compiler to flag all DO loop control variables that are declared in a parent block.

**LAXCTL or 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, then 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);
```

**LAXDCL or NOLAXDCL**

Specifying LAXDCL allows implicit declarations. NOLAXDCL disallows all implicit and contextual declarations except for BUILTINs and for files SYSIN and SYSPRINT.

**LAXDEF or 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).

**LAXENTRY or NOLAXENTRY**

Specifying LAXENTRY allows unprototyped entry declarations. Specifying NOLAXENTRY will cause the compiler to flag all unprototyped entry declarations, i.e. all ENTRY declares that do not specify a parameter list. Note
that this would mean that 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.

RULES(LAXENTRY) is the default.

**LAXIF or NOLAXIF**
Specifying RULES(NOLAXIF) will cause 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.

The following would all be flagged under NOLAXIF:
```
dcl i fixed bin;
dcl b bit(8);
...
if i then ...
if b then ...
```

**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 it thinks such a parameter has not been initialized.

**LAXLINK or 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, then under RULES(NOLAXLINK), the compiler would 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, then under RULES(NOLAXLINK), the compiler would flag an attempt to assign A3 to A4.

- the LINKAGE and other OPTIONS suboptions
  
  For example, if A5 is declared as ENTRY OPTIONS(ASM) and A6 as an ENTRY VARIABLE without the OPTIONS attribute, then under RULES(NOLAXLINK), the compiler would flag an attempt to assign A5 to A6 because the OPTIONS(ASM) in the declare of A5 implies that A5 has LINKAGE(SYSTEM) while A6 will have LINKAGE(OPTLINK) by default, since it has no OPTIONS attribute.

**LAXMARGINS or NOLAXMARGINS**
Specifying NOLAXMARGINS causes the compiler to flag, depending on the setting of the STRICT and XNUMERIC suboption, lines containing non-blank 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, then any statements that would be flagged because of the NOLAXMARGINS option will be reported as statement zero (since 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 the STRICT suboption, the compiler will flag any line containing non-blank characters after the right margin

**XNUMERIC**
Under the XNUMERIC suboption, the compiler will flag any line containing non-blank characters after the right margin except if the right margin is column 72 and columns 73 through 80 all contain numeric digits

**LAXNESTED | NOLAXNESTED**
Specifying RULES(NOLAXNESTED) causes the compiler to flag any executable code in a procedure that follows any subprocedures.

Specifying RULES(LAXNESTED) causes the compiler not to flag the executable code in a procedure that follows any subprocedures.

**LAXPUNC or NOLAXPUNC**
Specifying NOLAXPUNC causes the compiler to flag with an E-level message any place where it assumes punctuation that is missing.

For instance, given the statement ‘I = (1 * (2);’’, the compiler assumes that a closing right parenthesis was meant before the semicolon. Under RULES(NOLAXPUNC), this statement would be flagged with an E-level message; otherwise it would be flagged with a W-level message.

**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 */
```

**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:
Compile-time options

- A RETURN statement with an expression in a procedure that is coded without the RETURN option
- A RETURN statement without an expression in a procedure that is coded with the RETURN option

RULES(LAXRETURN) is the default.

LAXSCALE | NOLAXSCALE
Specifying NOLAXSCALE causes the compiler to flag any FIXED BIN(p,q) or FIXED DEC(p,q) declare where q < 0 or p < q.

It also causes the compiler to flag ROUND(x, p) when p < 0.

The message issued when flagging ROUND(x, p) is different from that issued when flagging 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.

LAXSEMI or NOLAXSEMI
Specifying NOLAXSEMI causes the compiler to flag any semicolons appearing inside comments.

LAXSTG or NOLAXSTG
Specifying NOLAXSTG causes the compiler to flag declares 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.

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 (since some customers declare B with placeholder attributes that do not describe the actual argument). For those customers whose parameter and argument declares match (or should match), specifying RULES(NOLAXSTG) may help detect more storage overlay problems.

LAXSTRZ or 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).

MULTICLOSE or NOMULTICLOSE
NOMULTICLOSE causes the compiler to flag all statements that force the closure of multiple groups of statement with an E-level message.

PADDING | NOPADDING
Specifying NOPADDING causes the compiler to flag all structures that contain padding.

PROCENDONLY | NOPROCENDONLY
Specifying NOPROCENDONLY causes any END statement closing 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.

RECURSIVE | NORECURSIVE
Specifying NORECURSIVE causes the compiler not to flag the use of the RECURSIVE attribute or any procedure that directly calls itself.

Specifying RECURSIVE causes the compiler to flag any use of the RECURSIVE attribute or any procedure that directly calls itself.

Note: Do not use RULES(NORECURSIVE) and DFT(RECURSIVE) together.
SELFASSIGN|NOSELFASSIGN
Specifying NOSELFASSIGN causes the compiler to flag all assignments where the source and the target are the same.
RULES(SELFASSIGN) is the default.

STOP|NOSTOP
Specifying NOSTOP causes all STOP and EXIT statements to be flagged.

UNREF or NOUNREF
Specifying NOUNREF causes the compiler to flag any level-1 AUTOMATIC variable which is not referenced and which, if it is a structure or union, contains no subelement which is referenced.

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.

Default: RULES (IBM BYNAME CONTROLLED NODECSIZE EVENDEC ELSEIF GOTO GLOBALDO NOLAXBIF NOLAXCTL LAXDCL NOLAXDEF LAXENTRY LAXIF LAXINOUT LAXLINK LAXNESTED LAXPUNC LAXMARGINS LAXQUAL LAXRETURN LAXSCALE LAXSEMI LAXSTG NOLAXSTRZ MULTICLOSE PADDING PROCENDONLY RECURSIVE SELFASSIGN STOP UNREF)
Compile-time options

SEMANTIC

This option specifies that the execution of the compiler’s semantic checking stage depends on the severity of messages issued prior to 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.
SNAP

This option specifies whether SNAP and PLIDUMP traceback output must be complete if an exception occurs.

SNAP

SNAP and PLIDUMP traceback output always includes all PL/I routines on the call stack. SNAP can significantly increase the size and reduce the performance of your programs. It is not recommended for production programs.

NOSNAP

SNAP and PLIDUMP traceback output may not be complete.
Compile-time options

**SOSI**

This option specifies how source containing the DBCS shift-out and shift-in characters is handled.

**NOSOSI**

Under the NOSOSI option, there will be no special handling of these characters.

**SOSI**

Under the SOSI option, the compiler will accept and correctly process source containing the DBCS shift-out and shift-in characters.
The SOURCE option specifies that a listing of the source input to the compiler be created.

ABBREVIATIONS: S, NS

SOURCE
   The compiler produces a listing of the source.

NOSOURCE
   The compiler does not produce a source listing.

A source listing is not produced unless syntax checking is performed.
Compile-time options

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 insure that they will be in the generated object module.
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.

Specifying NOSTMT implies NUMBER.

When the STMT option is specified, the source listing will include both the logical statement numbers and the source file numbers.
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
SYNTAX

This option specifies that the execution of the compiler’s syntax checking stage depends on the severity of messages issued prior to this stage of processing.

SYNTAX

No syntax checking is performed.

NOSYNTAX(S)

No syntax checking is performed if a severe error or unrecoverable error has been detected.

NOSYNTAX(E)

No syntax checking is performed if an error, severe error, or unrecoverable error has been detected.

NOSYNTAX(W)

No syntax checking is performed if a warning, error, severe error, or unrecoverable error has been detected.

If the NOSYNTAX option terminates the compilation, the cross-reference listing, attribute listing, source listing, and other listings that follow the source program are not produced.
SYSPARM

This option allows you to specify the value of the string that is returned by the macro facility built-in function SYSPARM.

\[
\text{SYSPARM} - \langle - \text{\textit{string}} - \rangle
\]

\textbf{string}

This string can be up to 64 characters long. A null string is the default, however, if you choose to specify a string value, see the note on \textit{strings} in step 2 on page 30 under “Rules for using compile-time options”.

For more information about the macro facility, see the \textit{PL/I Language Reference}.

DEFAULT: SYSPARM("")
SYSTEM

This option specifies the operating system and hardware platform under which the PL/I program will run. It also enforces the parameters that can be received by a MAIN procedure.

In addition, a suboption allows you to exploit the hardware platform on which the object code will run.

\[
\text{SYSTEM} - (\text{WINDOWS}) - \text{CICS} - \text{IMS}
\]

WINDOWS
  Specifies that the program runs under WINDOWS.

CICS
  Specifies that the program runs under CICS.

IMS
  Specifies that the program will run under IMS.

S486
  The object code is intended to run on a machine which has an 80486 or compatible chip. The code runs on machines with a Pentium chip, but not a 386 chip.

Pentium
  The object code is intended to run on a machine with a Pentium chip. The code does not run on machines without a Pentium chip.

For MAIN procedures compiled with SYSTEM(CICS), OPTIONS (BYVALUE) is assumed and PROCEDURE OPTIONS(BYADDR), if specified, is diagnosed.
Compile-time options

**TERMINAL**

This option determines whether or not diagnostic and information messages produced during compilation are displayed on the terminal.

TERMINAL

Messages are displayed on the terminal.

NOTERMINAL

No information or diagnostic compiler messages are displayed on the terminal.

ABBREVIATIONS: TERM, NTERM
The TEST option specifies the level of testing capability generated as part of the object code. It allows you to control the location of test hooks and to control whether or not the symbol table is generated.

The TEST option implies 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.

**NOTEST**
- Suppresses the generation of all testing information.

**TEST**
- Specifies that testing information should be included in the object code.
The USAGE option lets you choose IBM or ANS 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 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 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 would return a string whose length is z.

The SUBSTR(LOOSE) suboption may 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.
The WIDECHAR option specifies the format in which WIDECHAR data will be stored.

- **BIGENDIAN**
  Indicates that WIDECHAR data will be stored in bigendian 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 littleendian 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 bigendian 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.

DEFAULT: WIDECHAR( LITTLEENDIAN )
WINDOW

The WINDOW option sets the value for the \( w \) window argument used in various date-related built-in functions.

\[
\text{WINDOW}(w)
\]

The value for \( w \) is either an unsigned integer that represents the start of a fixed window or a negative integer that specifies a "sliding" window. For example, \( \text{Window}(-20) \) indicates a window that starts 20 years prior to the year when the program runs.

DEFAULT: WINDOW(1950)
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, for the compilation unit, all:

- defined EXTERNAL procedures
- defined EXTERNAL variables
- statically referenced EXTERNAL routines and variables
- dynamically called fetched modules

This file is written to the same directory as the object deck and has the extension "def".

For instance, given the program:

```plaintext
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 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. See the appendix for more details on the format of the ADATA file.
Compile-time options

The ADATA file is generated in the same directory as the object file and has an extension of "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.

The ADATA file is generated in the same directory as the object file and has an extension of "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.

The ADATA file is generated in the same directory as the object file and has an extension of "adt".

**NOSYN**
No syntax information is generated to the ADATA file.

**XMI**
An XMI side-file is created. This XMI is not intended to be read or interpreted except by other tools.

This file is written to the same directory as the object deck and has the extension "xmi".

**NOXMI**
No XMI side-file is created.

**XML**
An XML side-file is created. This XML file includes:
- the file reference table for the compilation
- the block structure of the program compiled
- the messages produced during the compilation

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:

```xml
<?xml encoding="UTF-8"?>
<!ELEMENT PACKAGE ((PROCEDURE)*, (MESSAGE)*, FILEREFERENCETABLE)>  
<!ELEMENT PROCEDURE (BLOCKFILE, BLOCKLINE, (PROCEDURE)*, (BEGINBLOCK)*)>  
<!ELEMENT BEGINBLOCK (BLOCKFILE, BLOCKLINE, (PROCEDURE)*, (BEGINBLOCK)*)>  
<!ELEMENT MESSAGE (MSNUMBER, MSGLINE?, MSGFILE?, MSGTEXT)>  
<!ELEMENT FILE (FILENAME, INCLUDEDFROMFILE?, INCLUDEDONLINE?, FILENAME)>  
<!ELEMENT FILEREFERENCETABLE (FILECOUNT, FILE*)>  
<!ELEMENT BLOCKFILE (#PCDATA)>  
<!ELEMENT BLOCKLINE (#PCDATA)>  
<!ELEMENT MSGNUMBER (#PCDATA)>  
<!ELEMENT MSGFILE (#PCDATA)>  
<!ELEMENT MSGLINE (#PCDATA)>  
<!ELEMENT MSGTEXT (#PCDATA)>  
<!ELEMENT FILECOUNT (#PCDATA)>  
```
Compile-time options

NOXML
No XML side-file is created.
Compile-time options

XML

The XML option lets you choose the case of the names in the XML generated by the XMLCHAR built-in function

```
XML { CASE ( UPPER | ASIS ) }
```

**CASE( UPPER | ASIS )**

Under the CASE(UPPER) suboption, the names in the XML generated by the XMLCHAR built-in function will all be in upper case.

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 declares. Note that if you the MACRO preprocessor without using the macro preprocessor option CASE(ASIS), then the source seen by the compiler will have all the names in upper case - and that would make specifying the XML(CASE(ASIS)) option useless.
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

NOXREF
Indicates that the compiler should not produce this information as part of the listing.

XREF
Specifies that the compiler should produce a cross-reference list.

In addition to the cross-reference list, the compiler produces a listing of unreferenced identifiers. In this list, variables do not appear if they are named constants or static nonassignable variables. If any field in a union or structure is referenced, the name of the union or structure does not appear. Level 1 names for unions or structures appear only if none of the members are referenced.

For an example and description of the content of the cross-reference table, see "Using the compiler listing” on page 175. If both XREF and ATTRIBUTES are specified, the two listings are combined.
Chapter 4. PL/I preprocessors

The PL/I compiler allows you to select one or more of the integrated preprocessors as required for use in your program. You can select the include preprocessor, macro facility, the SQL preprocessor, or the CICS preprocessor and the order in which you would like them to be called.

- The include preprocessor processes special include directives and incorporates external source files.
- The macro facility, 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 to allow you 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.

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:

```
PP(INCLUDE('ID(<directive>)'))
```

**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.

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)
```

**Examples**

This first example causes all lines that start with -INC (and possibly preceding blanks) to be treated as include directives:

```
pp( include('id(-inc)'))
```

This second example causes all lines that start with ++INCLUDE (and possibly preceding blanks) to be treated as include directives:

```
pp( include('id(++include)'))
```
Include preprocessor

Include preprocessor options environment variable
You can set the default options for the include preprocessor by using the IBM.PPINCLUDE environment variable. See "IBM.PPINCLUDE" on page 22.

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.

The macro preprocessing facilities of the compiler are described in the PL/I Language Reference manual.

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 from the list below.

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 quotes (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 and/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 may specify the options in any order.

Macro preprocessor options

The macro preprocessor supports the following options:

**CASE**
This option specifies if the preprocessor should convert the input text to uppercase.

```
CASE - (ASIS, UPPER)
```

**ASIS**
the input text is left "as is".

**UPPER**
the input text is to be converted to upper case.

**FIXED**
This option specifies the default base for FIXED variables.

```
FIXED - (DECIMAL, BINARY)
```

**DECIMAL**
FIXED variables will have the attributes REAL FIXED DEC(5).
Macro facility

**BINARY**
FIXED variables will have the attributes REAL SIGNED FIXED BIN(31).

**INCONLY**
This option specifies that the preprocessor should process only %INCLUDE and %XINCLUDE statements. When this option is in effect, you may use neither INCLUDE or XINCLUDE as a macro

- procedure name
- statement label
- variable name

**NOINCONLY**
This option specifies that the preprocessor should process all preprocessor statements and not only %INCLUDE and %XINCLUDE statements. This option and the INCONLY option are mutually exclusive, and for compatibility, NOINCONLY is the default.

**NAMEPREFIX**
This option specifies that the names of preprocessor procedures and variables must start with the specified character.

```
**NAMEPREFIX**-(character)
```

The character should be specified "as is" and should not be enclosed in quotes.

**NONAMEPREFIX**
This option specifies that the names of preprocessor procedures and variables are not required to start with one particular character. NONAMEPREFIX is the default.

**RESCAN**
This option specifies how the preprocessor should handle the case of identifiers when rescanning text.

```
**RESCAN**-(ASIS  UPPER)
```

**ASIS**
rescans will not be case-sensitive.

**UPPER**
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 compiled with PP(MACRO('RESCAN(ASIS)')), in the second display statement, the value of text is replaced by eins, but no further replacement occurs since under RESCAN(ASIS), eins does not match the macro variable eins since the former is left asis while the latter is uppercased. Hence the following text would be generated
Macro facility

```plaintext
    DISPLAY( zwei );
    DISPLAY( eins );
    But when compiled with PP(MACRO('RESCAN(UPPER)')), in the second
display statement, the value of text is replaced by eins, but further
replacement does occur since under RESCAN(UPPER), eins does match the
macro variable eins since both are uppercased. Hence the following text would
be generated
    DISPLAY( zwei );
    DISPLAY( zwei );
    DISPLAY( zwei );
    In short: 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 facility options environment variables

You can set the default options for the macro facility by using the IBM.PPMACRO
environment variable. See "IBM.PPMACRO" on page 22.

SQL preprocessor

You can use dynamic and static EXEC SQL statements in PL/I applications. Before
you can take advantage of EXEC SQL support, you must have installed IBM DB2
Universal Database (herinafter referred to as DB2) for Windows.

Workstation PL/I products support most of the function in DB2 and increased
function will be added in each successive release. If you specify newer DB2
functions while using a downlevel DB2 product, warning messages are generated
and those newer options are ignored.

Programming and compilation considerations

You need to consider specific options when using the PL/I SQL preprocessor. The
PL/I for Windows SQL preprocessor does not support variables that have either
the NONNATIVE or the HEXADEC attributes.

When you have EXEC SQL statements in your PL/I source program, use the
PP(SQL) option to process those statements:

```
    pp(sql('option-string'))
```

In the preceding example, 'option-string' is a character string enclosed in quotes. For
example, `pp(sql('dbname(Sample)'))` tells the preprocessor to work with the
SAMPLE database.

If you are using EXEC SQL statements in your program, you must specify the SQL
library in addition to the other link libraries in the linking command, for example:

```
    ilink myprog.obj db2api.lib
```

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 looking 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.
If the source uses %INCLUDE or other macro statements, invoke the MACRO preprocessor before the SQL preprocessor.

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 using DBCS and SBCS characters. In particular, you can use DBCS characters in the source program in the 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(EBCDIC)
  - DFT(HEXADEC)
- Do not use DECLARE STATEMENT statements in SQL queries, because the PL/I preprocessor always ignores these statements.

**SQL Users**

You must have DB2 Universal Database for Windows installed and started before you can compile a program containing EXEC SQL statements. To find out how to install DB2, refer to database installation guide for the platform you are using.

You can start the database manager by issuing the following at a command prompt:

db2start

**SQL preprocessor options**

The following syntax diagram illustrates all of the options supported by the SQL preprocessor.
Abbreviations: DB, BLK, CRESO, DT, ISOL, ON, TW, D, ND, INS, COL, CON, DISC, SQLR, SYNC

**DBNAME**

Specifies the original or alias name of a database. This option directs the preprocessor to process the SQL statements against the specified database. If you omit this option or do not specify a database name, the preprocessor uses the default database if an implicit connect is enabled. The default database is specified by the environment variable DB2DBDFT. Further information is available in your DB2 documentation.

The preprocessor must have a database to work with or an error occurs.

**BLOCK**

Specifies the type of record blocking to be used and how ambiguous cursors are to be treated. The valid values for this option are:

**UNAMBIG**

Blocking occurs for read-only cursors, those that are not specified as FOR UPDATE OF, have no static DELETE WHERE CURRENT OF statements, and have no dynamic statements. Ambiguous cursors can be updated.

**ALL**

Blocking occurs for read-only cursors, those that are not specified as FOR UPDATE OF, and for which no static DELETE WHERE CURRENT OF statements are executed. Ambiguous and dynamic cursors are treated as read-only.

**NO**

No blocking is done on any cursors in the package. Ambiguous cursors can be updated.

**BIND or NOBIND**

Determines whether or not a bind file bindname is created. The bind file has an extension .BND and is saved either in the current directory or the directory specified by the IBM_BIND environment variable. If you do not specify a bindname, the name defaults to the name of the input source file.

**CALL_RESOLUTION**

Determines whether the CALL statement will be executed as an invocation of
the deprecated sqleproc() API or as a normal SQL statement. Note that SQL0204 will be issued if the precompiler fails to resolve the procedure on a CALL statement with CALL_RESOLUTION IMMEDIATE.

**IMMEDIATE**

The CALL statement will be executed as a normal SQL statement. This is the default.

**DEFERRED**

The CALL statement will be executed as an invocation of the deprecated sqleproc() API.

**COLLECTION**

Specifies an eight character collection identifier for the package.

**schema-name**

Eight character identifier.

There is no default value for the COLLECTION option. If the COLLECTION is specified, a schema-name must also be provided.

**CONNECT**

Specifies the type of CONNECT that is made to the database.

1. Specifies that a CONNECT command is processed as a type 1 CONNECT. This is the default setting.

2. Specifies that a CONNECT command is processed as a type 2 CONNECT.

The default option value is CONNECT(1). The following option strings evaluate to CONNECT(1): CON, CONNECT, CON(), and CONNECT().

**DATETIME**

Determines the date and time format used when date and time fields are assigned to string representations in host variables. The following three-letter abbreviations are valid for the variable location:

**DEF**

Use the date/time format associated with the country code of the database. This is also the default if DATETIME is not specified.

**USA**

IBM standard for U.S. form.

Date format: mm/dd/yyyy

Time format: hh:mm xM (AM or PM)

**EUR**

IBM standard for European form.

Date format: dd.mm.yyyy

Time format: hh.mm.ss

**ISO**

International Standards Organization.

Date format: yyyy-mm-dd

Time format: hh.mm.ss

**JIS**

Japanese Industrial Standards.

Date format: yyyy-mm-dd

Time format: hh:mm:ss

**LOC**

Local form, not necessarily equal to DEF

**DBUID and DBPWD**

Allows you to specify a *userid* and *password* for those database managers which require that these values be supplied when a remote connection is attempted. For example, these values might be required during a compile against a remote database resident on a Windows server.

The options DBUID and DBPWD can be in either case, but the values of *userid* (maximum length is 8 characters) and *password* (maximum length is 18 characters) are case sensitive.
SQL preprocessor

The userid and password are only used by the SQL preprocessor to connect to the database manager during the compile process. When the application connects during execution, the userid and password for that connect must be provided on the EXEC SQL CONNECT statement in the program.

**DECK or NODECK**
This option specifies that the SQL preprocessor output source is written to a file with the extension .DEK and the file is put the current directory.

**DEPRECATE**
This 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 option is: DEPRECATE(STMT())

**DISCONNECT**
Specifies the type of DISCONNECT that is made to the database.

**EXPLICIT**
Specifies that only database connections that have been explicitly marked for release by the RELEASE statement are disconnected at commit. This is the default setting.

**AUTOMATIC**
Specifies that all database connections are disconnected at commit.

**CONDITIONAL**
Specifies that the database connections that have been marked RELEASE or have no open WITH HOLD cursors are disconnected at commit.

The default option value is DISCONNECT(EXPLICIT). The following option strings evaluate to DISCONNECT(EXPLICIT): DISC, DISCONNECT, DISC(), DISCONNECT().

**INCONLY or NOINCONLY**
Determines whether or not the SQL preprocessor should process only EXEC SQL INCLUDE statements.

When INCONLY is specified, no code is generated by the SQL preprocessor and all of the EXEC SQL INCLUDE statements are expanded. When INCONLY is specified, other options cannot be specified.

When NOINCONLY is specified, the SQL preprocessor processes all statements and generates code. INCONLY and NOINCONLY are mutually exclusive, and for compatibility, NOINCONLY is the default.
SQL preprocessor

**INSERT**
Requests that the data inserts be buffered to increase performance on the DB2/6000 Parallel Edition server.

**DEF**  Use standard INSERT with VALUES execution. This is the default setting.

**BUF**  Use buffering when executing INSERTs with VALUES.

*Note:* This option can only be used when precompiling against a DB2 Parallel Edition server. If INSERT is used against a DB2 V1.x server, it is ignored and a warning message is issued. If INSERT is used against a DB2 V2.x server, it is ignored, a warning message is issued, and the option is added to the bind file.

**ISOLATION**
Determines how far a program bound to this package can be isolated from the effect of other executing programs.

**CS**  Specifies Cursor Stability as the isolation level.

**RR**  Repeatable read

Specifies Repeatable Read as the isolation level.

**RS**  Specifies Read Stability as the isolation level. Read Stability ensures that the execution of SQL statements in the package is isolated from other application processes for rows read and changed by the application.

**UR**  Specifies Uncommitted Read as the isolation level.

**ONEPASS or TWOPASS**
ONEPASS is the default and indicates that host variables must be declared before use. Use of TWOPASS indicates that host variables do not need to be declared before use.

**OWNER**
Designates a 30-character *authorization-id* for the package owner. The owner must have the privileges required to execute the SQL statements contained in the package. Only a user with SYSADM or DBADM authority can specify an *authorization-id* other than the user ID. The default value is the primary *authorization-id* of the precompile/bind process. SYSIBM, SYSCAT, and SYSSTAT are not valid values for this option.

**PLAN, NOPLAN, or NOPLANSYNTAX**
Determines whether or not an access plan *planname* is created. If you do not specify a planname, the name defaults to the name of the input source file.

If you specify NOPLANSYNTAX, no plan is created and a syntax check is performed against DB2 Version 2.1 syntax.

**QUALIFIER**
Provides a 30-character implicit *qualifier-name* for unqualified objects contained in the package. The default is the owner’s authorization ID, whether or not owner is explicitly specified.

**SQLFLAG**
Identifies and reports on deviations from SQL language syntax specified in this option. If this option is not specified, the flagger function is not invoked. Further information is available in your DB2 documentation.

**MVSDB2V23**
SQL statements are checked against the MVS DB2 V2.3 SQL language syntax. This is the default setting.
SQL preprocessor

**MVSDB2V31**
SQL statements are checked against the MVS DB2 V3.1 SQL language syntax.

**MVSDB2V41**
SQL statements are checked against the MVS DB2 V4.1 SQL language syntax.

**SQLRULES**
Specifies whether type 2 CONNECTs should be processed according to the DB2 rules or the Standard (STD) rules based on ISO/ANS SQL92.

**DB2**
Allows the use of the SQL CONNECT statement to switch the current connection to another established (dormant) connection. This is the default setting.

**STD**
 Allows the use of the SQL CONNECT statement to establish a new connection only. The SQL SET CONNECTION must be used to switch to a dormant connection.

The default option value is SQLRULES(DB2). The following option strings evaluate to SQLRULES(DB2): SQLR, SQLRULES, SQLR(), SQLRULES().

**SYNCPOINT**
Specifies how commits or rollbacks are coordinated among multiple database connections.

**ONEPHASE**
Specifies that no Transaction Manager (TM) is used to perform a two-phase commit. A one-phase commit is used to commit the work done by each database in multiple database transactions. This is the default setting.

**TWOPHASE**
Specifies that the TM is required to coordinate two-phase commits among those databases that support this protocol.

**NONE**
Specifies that no TM is used to perform a two-phase commit, and does not enforce single updater, multiple reader. A COMMIT is sent to each participating database. The application is responsible for recovery if any of the commits fail.

The default option value is SYNCPOINT(ONEPHASE). The following option strings evaluate to SYNCPOINT(ONEPHASE): SYNC, SYNCPOINT, SYNC(), SYNCPOINT().

**TBQUAL**
Provides an 8-character implicit table-qualifier-name for unqualified objects contained in the package.

**VALIDATE**
Determines when the database manager checks for authorization errors and object not found errors. The package owner authorization ID is used for validity checking.

**BIND**
Validation is performed at precompile/bind time. If all objects do not exist, or all authority is not held, error messages are produced. If sqlerror
**SQL preprocessor**

`continue` is specified, a package/bind file is produced despite the error message, but the statements in error are not executable.

**RUN**

Validation is attempted at bind time. If all objects exist, and all authority is held, no further checking is performed at execution time. If all objects do not exist, or all authority is not held at precompile/bind time, warning messages are produced, and the package is successfully bound, regardless of the `sqlerror continue` option setting. However, authority checking and existence checking for SQL statements that failed these checks during the precompile/bind process can be redone at execution time.

**VERSION**

Defines the version identifier for a package. If this option is not specified, the package version will be “” (the empty string).

`version-id`

Specifies a version identifier that is any alphanumeric value, $, #, @, _, -, or ., up to 64 characters in length.

**AUTO**

The version identifier will be generated from the consistency token. If the consistency token is a timestamp (it will be if the LEVEL option is not specified), the timestamp is converted into ISO character format and is used as the version identifier.

---

**SQL preprocessor options environment variable**

You can set the default options for the SQL Preprocessor by using the IBM.PPSQL environment variable. See "IBM.PPSQL" on page 22.

**SQL preprocessor BIND environment variables**

If the BIND option is specified, the SQL preprocessor creates a bind file in the current directory for the program you compile. You can change the destination of the output file by setting the IBM.BIND environment variable, for example:

```
set ibm.bind=C:\bindlib
```

The SQL bind output file has the same name as the primary input file, unless otherwise specified, and an extension of BND.

---

**Coding SQL statements in PL/I applications**

You can code SQL statements in your PL/I applications using the language defined in *SQL Reference, Volume 1 and Volume 2*. Specific requirements for your SQL code are described in the sections that follow.

**Defining the SQL communications area**

A PL/I program that contains SQL statements must include an SQL communications area (SQLCA) As shown in Figure 1 on page 159 part of an SQLCA consists of an SQLCODE variable and an SQLSTATE variable.

- The SQLCODE value is set by the Database Manager 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 analyzing the result of an SQL statement. Like the SQLCODE variable, the SQLSTATE variable is set by the Database Manager after each SQL statement is executed.
The SQLCA should be included by using the SQL INCLUDE statement:

```
exec sql include sqlca;
```

The SQLCA must not be defined within an SQL declare section. The scope of the SQLCODE and SQLSTATE declaration must include the scope of all SQL statements in the program.

```
Dcl
  1 Sqlca,
  2 sqlda   char(8),   /* Eyecatcher = 'SQLCA '*/
  2 sqlcabc fixed binary(31), /* SQLCA size in bytes = 136 */
  2 sqlcode fixed binary(31), /* SQL return code */
  2 sqlerrm char(70) var,   /* Error message tokens */
  2 sqlerrp char(8),       /* Diagnostic information */
  2 sqlerrr(6) 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 sqlda,
    3 sqlwarn8 char(1),
    3 sqlwarn9 char(1),
    3 sqlwarna char(1),
    3 sqlstate char(5);   /* State corresponding to SQLCODE */
```

*Figure 1. The PL/I declaration of SQLCA*

### Defining SQL descriptor areas

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`

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. An SQLDA should be included by using the SQL INCLUDE statement:

```
exec sql include sqlda;
```

The SQLDA must not be defined within an SQL declare section.
Embedding SQL statements
The first statement of your PL/I program must be a PROCEDURE or a PACKAGE statement. You can add SQL statements to your program wherever executable statements can appear. 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 Department
    export Mgrno = :Mgr_Num
    where Deptno = :Int_Dept;
```

Comments
In addition to SQL statements, PL/I comments can be included in embedded SQL statements wherever a blank is allowed.

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
SQL statements or PL/I host variable declaration statements can be included by placing the following SQL statement at the point in the source code where the statements are to be embedded:
```
exec sql include member;
```

Margins
SQL statements must be coded in columns \( m \) through \( n \) where \( m \) and \( n \) are specified in the MARGINS\((m,n)\) compile-time option.

Names
Any valid PL/I variable name can be used for a host variable and is subject to the following restriction: Do not use host variable names, external entry names, or access plan names that begin with 'SQL', 'DSN', or 'IBM'. These names are reserved.
for the database manager or PL/I. The length of a host variable name must not exceed 100 characters.

**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"
- "Declaring host variables"

**Using arrays as host variables**

You can use an array as a host variable only when it is an array of indicator variables for a host structure. That array 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 preprocessor does not use the data attribute defaults specified in the PL/I DEFAULT statement. The preprocessor ignores the DEFINE ALIAS and DEFINE STRUCT statements, so variables with attributes that are dependent on them may not be used as host variables. 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:
SQL preprocessor

- A prefix operator applied to an expression where the expression can be collapsed to an integer
- An add or sub 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, LENGTH, and MAXLENGTH
- A number that is an integer

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 non-negative 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.
- 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**

```
SQL TYPE IS BINARY (length)
```

**VARBINARY**

```
SQL TYPE IS VARBINARY (length)
```
SQL preprocessor

Result set locator

```sql
SQL TYPE IS RESULT_SET_LOCATOR
```

ROWID

```sql
SQL TYPE IS ROWID
```

Table locator

```sql
SQL TYPE IS TABLE LIKE table-name AS LOCATOR
```

LOB file reference

```sql
SQL TYPE IS BLOB_FILE
```

LOB locator

```sql
SQL TYPE IS BLOB_LOCATOR
```

LOB variable

```sql
SQL TYPE IS BLOB (length)
```

BLOB

You can also use **BINARY LARGE OBJECT** as an alternative for BLOB.

CLOB

You can also use either **CHARACTER LARGE OBJECT** or **CHAR LARGE OBJECT** as an alternative for CLOB.

XML LOB variable

```sql
SQL TYPE IS XML AS BLOB (length)
```

BLOB

You can also use **BINARY LARGE OBJECT** as an alternative for BLOB.

CLOB

You can also use either **CHARACTER LARGE OBJECT** or **CHAR LARGE OBJECT** as an alternative for CLOB.

XML file reference
SQL preprocessor

```
SQL TYPE IS XML AS
BLOB_FILE
CLOB_FILE
DBCLOB_FILE
```

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 the
following table. If a host variable appears with an indicator variable, the SQLTYPE
is the base SQLTYPE plus one.

*Table 3. SQL data types generated from PL/I declarations*

<table>
<thead>
<tr>
<th>PL/I Data Type</th>
<th>SQLTYPE of Host Variable</th>
<th>SQLLEN of Host Variable</th>
<th>SQL Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN FIXED(n), n &lt; 16</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>BIN FIXED(n), n ranges from 16 to 31</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DEC FIXED(p,s)</td>
<td>484</td>
<td>p (byte 1) s</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>BIN FLOAT(p), 22 ≤ p ≤ 53</td>
<td>480</td>
<td>8</td>
<td>FLOAT</td>
</tr>
<tr>
<td>DEC FLOAT(m), 7 ≤ m ≤ 16</td>
<td>480</td>
<td>8</td>
<td>FLOAT</td>
</tr>
<tr>
<td>CHAR(n), 1 ≤ n ≤ 254</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHAR(n) VARYING, 1 ≤ n ≤ 4000</td>
<td>448</td>
<td>n</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>CHAR(n) VARYING, n &gt; 4000</td>
<td>456</td>
<td>n</td>
<td>LONG VARCHAR</td>
</tr>
<tr>
<td>GRAPHIC(n), 1 ≤ n ≤ 127</td>
<td>468</td>
<td>n</td>
<td>GRAPHIC(n)</td>
</tr>
<tr>
<td>GRAPHIC(n) VARYING, 1 ≤ n ≤ 2000</td>
<td>464</td>
<td>n</td>
<td>VARGRAPHIC(n)</td>
</tr>
<tr>
<td>GRAPHIC(n) VARYING, n &gt; 2000</td>
<td>472</td>
<td>n</td>
<td>LONG VARGRAPHIC</td>
</tr>
</tbody>
</table>

Since SQL does not have single or extended precision floating-point data type, if a
single or extended precision floating-point host variable is used to insert data, it is
converted to a double precision floating-point temporary and the value in the
temporary is inserted into the database. If the single or extended precision
floating-point host variable is used to retrieve data, a double precision
floating-point temporary is used to retrieve data from the database and the result
in the temporary variable is assigned to the host variable.

The following table can be used to determine the PL/I data type that is equivalent
to a given SQL data type.

*Table 4. SQL data types mapped to PL/I declarations*

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>PL/I Equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>BIN FIXED(15)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 4. SQL data types mapped to PL/I declarations (continued)

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>PL/I Equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>BIN FIXED(31)</td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DEC FIXED(p) or</td>
<td>p = precision and s = scale; 1 ≤ p ≤ 31 and 0 ≤ s ≤ p</td>
</tr>
<tr>
<td></td>
<td>DEC FIXED(p,s)</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>BIN FLOAT(p) or</td>
<td>22 ≤ p ≤ 53; 7 ≤ m ≤ 16</td>
</tr>
<tr>
<td></td>
<td>DEC FLOAT(m)</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>1 ≤ n ≤ 254</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>CHAR(n) VAR</td>
<td>1 ≤ n ≤ 4000</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>CHAR(n) VAR</td>
<td>n &gt; 4000</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>GRAPHIC(n)</td>
<td>n is a positive integer from 1 to 127 that refers to the number of double-byte characters, not to the number of bytes</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>GRAPHIC(n) VAR</td>
<td>n is a positive integer that refers to the number of double-byte characters, not to the number of bytes; 1 ≤ n ≤ 2000</td>
</tr>
<tr>
<td>LONG VARGRAPHIC</td>
<td>GRAPHIC(n) VAR</td>
<td>n &gt; 2000</td>
</tr>
<tr>
<td>DATE</td>
<td>CHAR(n)</td>
<td>n must be at least 10</td>
</tr>
<tr>
<td>TIME</td>
<td>CHAR(n)</td>
<td>n must be at least 8</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>CHAR(n)</td>
<td>n must be at least 26</td>
</tr>
</tbody>
</table>

### Determining compatibility of SQL and PL/I data types

PL/I host variables in SQL statements must be type compatible with the columns which 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 from 22 to 53, or DEC FLOAT(m) where m is from 7 to 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.
- Graphic data types are compatible with each other. A GRAPHIC or VARGRAPHIC column is compatible with a fixed-length or varying-length PL/I graphic 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. For example:

```pli
  dcl 1 A,
       2 B,
       3 C1 char(...),
       3 C2 char(...);
```
In this example, B is the name of a host structure consisting of the scalars C1 and C2.

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 161:
- CHARACTER, GRAPHIC, or WIDECHAR
- FIXED BIN, FIXED DEC, or FLOAT
- SQL TYPE

**Using indicator variables**

An indicator variable is a 2-byte integer (BIN FIXED(15)). 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.

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).

Given the statement:

```sql
exec sql
fetch Cls_Cursor into :Cls_Cd,
 :Day :Day_Ind,
 :Bgn :Bgn_Ind,
 :End :End_Ind;
```

Variables can be declared as follows:

```sql
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**

The following example shows the declaration of a host structure and an indicator array followed by an SQL statements that could be used to retrieve the data into the host structure.

```pli
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;
```
Using statements and compile-time options
This topic describes the way the preprocessor handles statements and compile-time
options with SQL statements.

CONNECT TO statement:
You can use a host variable to represent the database name you want your
application to connect to, for example:
exec sql connect to :dbase;

If a host variable is specified:
• It must be a character or a character varying variable.
• It must be preceded by a colon and must not be followed by an indicator
  variable.
• The server-name that is contained within the host variable must be left-justified.
• If the length of the server name is less than the length of the fixed-length
  character host variable, it must be padded on the right with blanks.
  dcl dbase char(10);
  dbase = 'SAMPLE'; /* blanks are padded automatically */
  exec sql connect to :dbase;
• If a varying character host variable is used, you may receive the following
  warning from the compiler. You can ignore this message.
  IBM1214I W xxx.x A dummy argument is created for argument
  number 6 in entry reference SQLESTRD_API

DECLARE TABLE statement:
The preprocessor ignores all DECLARE TABLE statements.

DECLARE STATEMENT statement:
The preprocessor ignores all DECLARE STATEMENT statements.

Large Object (LOB) support
Binary Large Objects (BLOBs), Character Large Objects (CLOBs), and Double Byte
Character Large Objects (DBCLOBs), along with the concepts of LOB LOCATORS
and LOB FILES are recognized by the preprocessor. See the DB2 manuals for more
information on these subjects.

LOBS, CLOBS, and BLOBS can be as large as 2,147,483,640 bytes long (2 Gigabytes
- 8 bytes for PL/I overhead). Double Byte CLOBS can be 1,073,741,820 characters
long (1 Gigabyte - 4 characters for PL/I overhead).

Note: Under CMPAT(V1), the size of a BLOB, CLOB, or DBCLOB must not exceed
32767 KB.

Sample programs for LOB support
Three sample programs are provided to show how LOB types can be used in PL/I
programs:
SQLLOB1.PLI
  Shows how to fetch a BLOB from the database into a file.
SQL preprocessor

**SQLLOB2A.PLI**
Shows how to use LOCATOR variables to modify a LOB without any movement of bytes until the final assignment of the LOB expression.

**SQLLOB2B.PLI**
Fetches the CLOB created in SQLLOB2A.PLI into a file for viewing.

**User defined functions sample programs**
You must install the following items to access the User Defined Function (UDF) sample programs:
- DB2 V2.1 or later
- Sample database

Several PL/I programs have been included to show how to code and use UDFs. Here is a short description of how to use them.

The file UDFDLL.PLI contains five sample UDFs. While these are simple in nature, they show basic concepts of UDFs.

- **MyAdd**
  Adds two integers and returns the result in a third integer.

- **MyDiv**
  Divides two integers and returns the result in a third integer.

- **MyUpper**
  Changes all lowercase occurrences of a,e,i,o,u to uppercase.

- **MyCount**
  Simple implementation of counter function using a scratchpad.

- **ClobUpper**
  Changes all lowercase occurrences of a,e,i,o,u in a CLOB to uppercase then writes them out to a file.

Use the command file bldudfdll to compile and link it into the udfdll library.

After the udfdll library has been compiled and linked, copy it to the user defined function directory for your database instance.

Before the functions can be used they must be defined to DB2. This is done using the CREATE FUNCTION command. The sample program, addudf.pli, has been provided to perform the CREATE FUNCTION calls for each UDF. CREATE FUNCTION calls would look something like the following:

```
CREATE FUNCTION MyAdd ( INT, INT ) RETURNS INT NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyAdd'

CREATE FUNCTION MyDiv ( INT, INT ) RETURNS INT NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyDiv'

CREATE FUNCTION MyUpper ( VARCHAR(61) ) RETURNS VARCHAR(61) NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyUpper'

CREATE FUNCTION MyCount ( ) RETURNS INT NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyCount'
```

SCRATCHPAD
CREATE FUNCTION ClobUpper ( CLOB(5K) ) RETURNS CLOB(5K) NO SQL 
LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER 
STYLE DB2SQL EXTERNAL NAME 'udfdll!ClobUpper'

These are just sample CREATE FUNCTION commands. Consult your DB2 manuals 
for more information or refinement.

Use the command file b1daddudf to compile and link the addudf.pli program. 
After it is compiled and linked, run it to define the user defined functions to your 
database.

Several sample PL/I programs are provided that call the user defined functions 
you have just created and added to the database:

**UDFMYADD.PLI**
Fetched ID and Dept from the STAFF table then adds them together by 
calling MyAdd UDF. Use the command file b1dmyadd to compile and link it.

**UDFMYDIV.PLI**
Fetched ID and Dept from the STAFF table then divides them by calling 
MyDiv UDF. Use the command file b1dmydiv to compile and link it.

**UDFMYUP.PLI**
Fetched Name from the STAFF table then calls MyUpper to change the vowels 
to uppercase. Use the command file b1dmyup to compile and link it.

**UDFMYCNT.PLI**
Fetched ID from the STAFF table, outputs the count of the call, then divides 
ID by the count. Use the command file b1dmycnt to compile and link it.

**UDFCLOB.PLI**
Fetched the resume for employee '000150' then calls ClobUpper to change 
the vowels to uppercase. Use the command file b1dclobu to compile and link it. After this program is run, look in the file udfclob.txt for the results.

Once these sample programs are compiled, linked, and the UDFs defined to DB2, 
the PL/I programs can be run from the command line.

These UDFs may also be called from the DB2 Command Line just line any other builtin DB2 function. For further information on how to customize and get the most out of your UDFs, please refer to your DB2 manuals.

**Handling SQL error return codes**

PL/I provides a sample program DSNTIAR.PLI that you can use to translate an 
SQLCODE into a multi-line message for display purposes. This PL/I program 
provides the same function as the DSNTIAR program on mainframe DB2*.

You must compile DSNTIAR with the same DEFAULT and SYSTEM compile time 
options that are used to compile the programs that use DSNTIAR.

You must compile DSNTIAR with the following compiler options:
• DEFAULT(ASCII NATIVE LINKAGE(OPTLINK))
• SYSTEM(WINDOWS)

The caller must declare the entry and conform to the interface as described in the mainframe DB2 publications. For your information, the declaration is of the following form:
SQL preprocessor

dcl dsntiar entry options(asm inter retcode);

Three arguments are always passed:

arg 1
This input argument must be the SQLCA.

arg 2
This input/output argument is a structure of the form:

dcl 1 Message,
    2 Buffer_length fixed bin(15) init(n), /* input */
    2 User_buffer char(n); /* output */

You must fill in the appropriate value for n.

arg 3
This input argument is a FIXED BIN(31) value that specifies logical record length.

SQL compatibility and migration considerations

The workstation compilers tolerate the following statement:

' EXEC SQL CONNECT :userid IDENTIFIED BY :passwd'

The preceding statement is translated by the PL/I SQL preprocessor and sent to the database precompiler services as:

' EXEC SQL CONNECT'

The allows VM SQL/DS users to compile their programs without making significant changes.

CICS support

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. Without invoking the CICS preprocessor, real code cannot be generated.

You can use EXEC CICS statements in PL/I applications that run as transactions under CICS.

You can develop these applications under CICS on Windows for eventual execution under CICS that particular development platform or under CICS/ESA, CICS/MVS, or CICS/VSE systems on S/390.

Make sure that the CICS installation adds all the OPT\... settings to your system environment variables for Windows support. It is not necessary that the CICS system be operational when you are compiling your programs.

Programming and compilation considerations

When you are developing programs for execution under CICS,

• You must use the SYSTEM(CICS) compile-time option.
• The MACRO option must follow the CICS option of PP.
If your CICS programs include files or use macros that contain EXEC CICS statements, you must also use either the MACRO compile-time option or the MACRO option of PP before the CICS option of the PP option as shown in the following example:

```
pp (macro(...) cics(...) macro(...) )
```

Make sure that INC is specified as an extension on the INCLUDE(EXT) compile-time option, see “INCLUDE” on page 66.

The IBM.SYSLIB or INCLUDE environment variable must specify the CICS include file directories, for example:

```
set include=d:\cicsnnn\plihdr;
```

The PL/I declarations generated by the CICSMAP, the Basic Mapping Support (BMS) utility, are placed in the first directory specified in the INCLUDE environment variable. For more information, see “Setting compile-time environment variables” on page 21.

Output produced in one of the following ways is written to the CPLI transient data queue (TDQ):
- PUT statements to SYSPRINT
- Messages written to the MSGFILE
- DISPLAY statements

Output produced by PLIDUMP is always written to the CPLD transient data queue.

The full workstation CICS API is supported for PL/I programs. Support is also provided for PL/I programs to use:
- External Presentation Interface (EPI)
- External Call Interface (ECI)
- External Transaction Initiation (ETI)

Other PL/I considerations that apply on S/390 CICS apply to CICS on the workstation also. The program behaves as though the STAE option is always in effect. The NOSTAE option is not supported.

If you are developing applications for eventual execution on S/390 CICS subsystems, you can check your PL/I programs for reentrancy violations with the DEFAULT(NONASSIGNABLE) compile-time option.

For compatibility with CICS/ESA, CICS/MVS, and CICS/VSE, make sure that the EXEC CICS commands are in upper case.

You can use PL/I FETCH and RELEASE under CICS.

A CICS program must not have more than one procedure that has OPTIONS(MAIN).

The EXEC CICS ADDRESS and other similar commands that return a pointer to a CICS control block (such as the TWA COMMAREA, and ACEE) might return a SYSNULL() pointer if the control block does not exist. (For example, '00000000'x not 'FF000000'x) Your programs must use the SYSNULL built-in function to test such pointers.
CICS support

Each PL/I compilation unit processed by the CICS preprocessor generates the following:

```plaintext
dcl IBM_CICS_ID char(n) static init('cics-id-and-version');
```

The name, version, and release level of the CICS system for which your program was compiled are indicated.

The CICS preprocessor requires the compiler option DEFAULT(ASCII NATIVE IEEE) be in effect.

You must have CICS installed before you can compile a program containing EXEC CICS statements. To find out how to install CICS on your workstation, refer to the installation instructions for that product.

CICS preprocessor options

The following syntax diagram show options supported by the CICS preprocessor.

```
PP(--CICS(--' --NOSOURCE --NOPRINT --NODECK --EDF
--SOURCE --PRINT --DECK --NOEDF

--DEBUG --NODEBUG

```

Abbreviations: S, NS, D, ND

**SOURCE or NOSOURCE**

Specifies whether or not the source input to the CICS preprocessor is printed.

**PRINT or NOPRINT**

Specifies whether or not the source code generated by the CICS preprocessor is printed in the source listing(s) produced by subsequent preprocessors or the compiler.

**DECK or NODECK**

Specifies that the CICS preprocessor output source is written to a file with the extension .DEK. The file is in the current directory.

**EDF or NOEDF**

Specifies whether or not the CICS Execution Diagnostic Facility (EDF) is to be enabled for the PL/I program. There is no performance advantage in specifying NOEDF, but the option can be useful in preventing CICS commands from appearing on EDF displays in well tested programs.

**DEBUG or NODEBUG**

Specifies whether or not the CICS preprocessor is to pass source program line numbers to CICS for use by the CICS Execution Diagnostic Facility (EDF).

CICS preprocessor options environment variables

You can set the default options for the CICS preprocessor by using the IBM.PPCICS environment variable. See "IBM.PPCICS" on page 22.
Coding CICS statements in PL/I applications

You can code CICS statements in your PL/I applications using the language defined in TXSeries for Multiplatforms, CICS Application Programming Guide in TXSeries for Multiplatforms Information Center at publib.boulder.ibm.com/infocenter/txformp/v7r1/.

Specific requirements for your CICS code are described in the sections that follow.

Embedding CICS statements

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 your program uses PL/I macros that generate EXEC CICS statements, you must use one of the following:

- 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

This section describes the rules and guidelines that apply to PL/I support of CICS on the workstation.

You can use PL/I with CICS facilities to write application programs (transactions) for CICS subsystems. If you do this, CICS provides facilities to the PL/I program that would normally be provided directly by the operating system. These facilities include most data management facilities and all job and task management facilities.

You should observe the following rules to ensure compatibility with S/390 PL/I CICS support.
CICS support

- Do not use macro level support, only command level support is provided.
- Do not use any PL/I input or output except:
  - Stream output for SYSPRINT
  - PLIDUMP

Since these are intended for debugging purposes only, you should not include them in production programs for performance reasons.
- Do not use the following statements:
  - DELAY
  - WAIT
- Do not use the PLISRTx built-in subroutines.
- Do not make calls to IMS using the PLITDLI, ASMTDLI, or EXEC DLI.
- Do not communicate with FORTRAN, COBOL, or C, using PL/I interlanguage facilities. However, CICS programs written in different languages can communicate with each other using EXEC CICS LINK or XCTL commands.

Subroutines written in a language other than PL/I can be called using PL/I interlanguage facilities providing those subroutines do not contain any EXEC CICS code. If you want to communicate with a non-PL/I program that contains EXEC CICS code, you must use EXEC CICS LINK or EXEC CICS XCTL as stated.

CICS abends used for PL/I programs

APLS

This abend is issued on termination, if termination is caused by the ERROR condition, and the ERROR condition was not caused by an abend (other than an ASRA abend).

This is the abend code issued by PL/I when either:
1. A transaction terminates in error due to a PL/I software interrupt (CONVERSION, for example), and there is no ERROR ON-unit
2. The program takes normal return from the ERROR ON-unit.

Because the program failed, the failure must be reflected to CICS on your workstation as an abend so that DTB, and so on, can occur if necessary.

APLT

An error was detected in the user exit.

CICS run-time user exit

It is strongly recommended that you review and modify (if necessary) the IBM-supplied CICS user exit, CEEFXITA. See “Using the CICS run-time user exit” on page 375.
Chapter 5. Compilation output

The results of compilation depend on how error-free your source program is and on the compile-time options you specify. Results can include diagnostic messages, a return code, and other output saved to disk (for example, an object module and a listing). The following section describes a sample compiler listing. "Compiler output files" on page 183 describes other kinds of output files you can request from the compiler.

Using the compiler listing

During compilation, the compiler generates listings that contain information about the source program, the compilation, and the object module. The TERMINAL option sends diagnostics and statistics to your terminal. The IBM.PRINT environment variable specifies the output directory for printable listing files (see "IBM.PRINT" on page 23 for more information on the IBM.PRINT environment variable). The following description of the listing refers to its appearance on a printed page.

This listing for CHIMES program highlights some of the more useful parts of the compiler listing. Figure 3 on page 176 is similar to the compiler listing for that program.
Using the compiler listing

Options Specified

Environment:

Command: op

Line.File Process Statements

1.0 *PROCESS MACRO S A(F) X AG;
2.0 *PROCESS NOT(\''\) OR(\'|');

Options Used

ADDEXT
+ AGGREGATE(DECIMAL)
+ ATTRIBUTES(FULL)
  BIFPREC(31)
  BLANK('09x')
  CASERULES( KEYWORD( MIXED ) )
  CHECK( NOCONFORMANCE NOSTORAGE )
  CMPAT(LE)
  CODEPAGE(00819)
  NOCOMPIL(E)
  NOCOPYRIGHT
  CURRENCY('$$')
  NODDLC
  DEFAULT(IBM ASSIGNABLE NOINITFILL NONCONNECTED LOWERINC
    DESCRIPTOR DESC LIST DUMMY(ALIGNED) ORDINAL(MIN)
    BYADDR RETURNS(BYVALUE) LINKAGE(OPTLINK) NORETCODE
    NOINLINE REORDER NOOVERLAP NONRECURSIVE ALIGNED NULLSYS
    BINARG PSEUDODUMMY NULLSTRADDR NULLSTRPTR(NULL) EVENDEC
    SHORT(HEXADEC ASCII IEEE NATIVE NATIVEADDR E(IEEE))
  DEPRECATED(
    BUILTIN()
    ENTRY()
    INCLUDE()
    VARIABLE()
    STMT()
  )
  DEPRECATENEXT(
    BUILTIN()
    ENTRY()
    INCLUDE()
    VARIABLE()
    STMT()
  )
  NODLLINIT
  NOEXIT
  EXTRN(SHORT)
  FLAG(W)
  FLOATINMATH(ASIS)
  NOGONUMBER
  NOGRAPHIC
  NOIGNORE
  IMPRECISE
  INCAFTER(PROCESS('"'))

CHIMES program compiler listing

Figure 3. CHIMES program compiler listing
Using the compiler listing

5724-B67 IBM(R) PL/I for Windows

Chapter 5. Compilation output
Using the compiler listing

Compiler Source

CHIMES: PROC OPTIONS(MAIN); /* Play a tune using DOSBEEP tones */
DCL { REST VALUE( 0 ), /* Declare Named Constants */
G4 VALUE( 392 ), /* for note and rest tone */
C5 VALUE( 523 ), /* values and timings. */
DS VALUE( 587 ),
ES VALUE( 657 ),
WHOLE VALUE( 800 ); } FIXED BIN(31);
DCL NOTES(19,2) STATIC NONASGN FIXED BIN(31)
INIT( ES, (WHOLE/2), /* Declare tone and timing */
C5, (WHOLE/2), /* for each note of tune. */
DS, (WHOLE/2),
G4, (WHOLE), /* Initial values may be */
REST, (WHOLE/2), /* restricted expressions */
G4, (WHOLE/2), /* using Named Constants */
DS, (WHOLE/2), /* previously defined in */
ES, (WHOLE/2), /* this program. */
C5, (WHOLE),
REST, (WHOLE/2),
ES, (WHOLE/2),
C5, (WHOLE/2),
DS, (WHOLE/2),
G4, (WHOLE),
REST, (WHOLE/2),
G4, (WHOLE/2),
DS, (WHOLE/2),
ES, (WHOLE/2),
Using the compiler listing

Chapter 5. Compilation output
Using the compiler listing

Options specified
This section of the compiler listing shows any compile-time options you specified. Options shown under Install: are specified in your IBM.OPTIONS environment variable. Options shown under Command: indicate that these options were specified on the command line when you invoked the compiler (there are no command options in this example). Options specified with the *PROCESS or %PROCESS statement are shown below the command options.

Options used
The compiler listing includes a list of all compile-time options used, including the default options. If an option is marked with a plus sign (+), the default has been changed. If any compile-time options contradict each other, the compiler uses the one with the highest priority. The following list shows which options the compiler uses, beginning with the highest priority:

- Options specified with the *PROCESS or %PROCESS statement.
- Options specified when you invoked the compiler with the PLI command.
- Install options installed either at installation time or by the IBM.OPTIONS environment variable (see "IBM.OPTIONS" on page 21 for more information on the IBM.OPTIONS environment variable).

Using the NUMBER option
The statement numbers shown are generated by the NUMBER option. In this case, the statement begins on the 14th line in file 1. The File Reference Table at the bottom of the listing also shows that file 1 refers to D:\ibmpli\samples\chimes.pli.

By generating these statement numbers during compilation, you can locate lines that need editing (indicated in messages, for example) without having to refer to the listing.

Attribute and cross-reference table
If you specify the ATTRIBUTES option, the compiler provides an attribute table containing a list of the identifiers in the source program together with their declared and default attributes in the compiler listing. The FULL attribute lists...
Using the compiler listing

all identifiers and attributes. If you specify the SHORT suboption for ATTRIBUTES, unreferenced identifiers are not listed.

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 Line/File number (the statement number inside the file and the file number, respectively) in which they appear in the compiler listing.

An identifier appears in the Sets: part of the cross-reference table if it is:
• The target of an assignment statement.
• Used as a loop control variable in DO loops.
• Used in the SET option of an ALLOCATE or LOCATE statement.
• Used in the REPLY option of a DISPLAY statement.

If there are unreferenced identifiers, they are displayed in a separate table (not shown in this example).

If you specify ATTRIBUTES and XREF (as in this example), the two tables are combined.

Explicitly-declared variables are listed with the number of the DECLARE statement in which they appear. Implicitly-declared variables are indicated by asterisks and contextually declared variables (HBOUND and LBOUND in this example) are indicated by plus (+) signs. (Undeclared variables are also listed in a diagnostic message.)

The attributes INTERNAL and REAL are never included; they can be assumed unless the respective conflicting attributes, EXTERNAL and COMPLEX, are listed.

For a file identifier, the attribute FILE always appears, and the attribute EXTERNAL appears if it applies; otherwise, only explicitly declared attributes are listed.

For an array, the dimension attribute is printed first. If the bound of an array is a restricted expression, the value of that expression is shown for the bound; otherwise an asterisk is shown.

If the length of a bit string or character string is a restricted expression, that value is shown, otherwise an asterisk is shown.

Aggregate length table

If you specified the AGGREGATE option, the compiler provides an aggregate length table in the compiler listing. The table shows how each aggregate in the program is mapped. Table 5 shows the headings for the aggregate length table columns and the description of each.

Table 5. Aggregate length table headings and description

<table>
<thead>
<tr>
<th>Heading</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line/File</td>
<td>The statement number and file number in which the aggregate is declared</td>
</tr>
<tr>
<td>Offset</td>
<td>The byte offset of each element from the beginning of the aggregate</td>
</tr>
<tr>
<td>Total Size</td>
<td>The total size in bytes of the aggregate</td>
</tr>
<tr>
<td>Base Size</td>
<td>The size in bytes of the data type</td>
</tr>
<tr>
<td>Identifier</td>
<td>The name of the aggregate and the element within the aggregate</td>
</tr>
</tbody>
</table>

File reference table

The Included From column of the File reference table indicates where the corresponding file from the Name column was included. The first entry in this
Using the compiler listing

column is blank because the first file listed is the source file. Entries in the
**Included From** column show the line number of the include statement
followed by a period and the file number of the source file containing the
include.

7 **Component, return code, diagnostic messages, time**
The last part of the compiler listing consists of the following headings:

**Component**
Shows you which component or processor is providing the information.
Either the macro facility, if invoked, or the compiler itself can provide you
with informational messages.

**Return code**
Shows you the highest return code generated by the component, issued
upon completion of compilation. Possible return codes are:

0 **(Informational)**
No warning messages detected (as in this example). The compiled
program should run correctly. The compiler might inform you of a
possible inefficiency in your code or some other condition of interest.

4 **(Warning)**
Indicates that the compiler found minor errors, but the compiler could
correct them. The compiled program should run correctly, but might
produce different results than expected or be significantly inefficient.

8 **(Error)**
Indicates that the compiler found significant errors, but the compiler
could correct them. The compiled program should run correctly, but
might produce different results than expected.

12 **(Severe error)**
Indicates that the compiler found errors that it could not correct. If the
program was compiled and an object module produced, it should not
be used.

16 **(Unrecoverable error)**
Indicates an error-forced termination of the compilation. An object
module was not successfully created.

Note: When coding CMD files for PL/I, you can use the return code to
decide whether or not post-compilation procedures are performed.

**Messages**
Indicates:
- The number of messages issued, if any
- The number of messages suppressed, if any, because they were equal to
  or below the severity level set by the FLAG compile-time option.

Messages for the compiler, macro facility, SQL preprocessor, and run-time
environment are listed and explained in *Messages and Codes*.

Only messages of the severity above that specified by the FLAG option are
issued. The messages, statements, and return code appear on your screen
unless you specify the NOTERMINAL compile-time option.

**Time**
Shows you the total time the component took to process your program.
### Compiler output files

If you compile a program using default options, an object module is created in the current directory. By altering compile-time options, you can request other output to be created in addition to the object module. Table 6 lists other possible compilation outputs which are also located in the current directory by default.

All compiler output files use the same file name as the main program file. The file extensions are specified in the following table.

*Table 6. Possible compilation disk outputs*

<table>
<thead>
<tr>
<th>Output</th>
<th>File extension</th>
<th>How requested (compile-time option)</th>
<th>How relocated (environment variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessed source text</td>
<td>DEK</td>
<td>DECK option of appropriate preprocessor</td>
<td>IBM.DECK</td>
</tr>
<tr>
<td>Object module</td>
<td>OBJ</td>
<td>OBJECT</td>
<td>IBM.OBJECT</td>
</tr>
<tr>
<td>Object listing</td>
<td>ASM</td>
<td>LIST</td>
<td>IBM.PRINT</td>
</tr>
<tr>
<td>Template .DEF file</td>
<td>DEF</td>
<td>XINFO(DEF)</td>
<td>IBM.OBJECT</td>
</tr>
<tr>
<td>Message listing</td>
<td>XML</td>
<td>XINFO(XML)</td>
<td>IBM.OBJECT</td>
</tr>
</tbody>
</table>

**Note:** You always receive a .LST file containing the program listing.
Compiler output files
Chapter 6. Linking your program

The following sections describe how to link object files produced by the compiler into either an executable program file (.EXE) or dynamic link library (.DLL).

Every .EXE that you build must contain exactly one main routine, that is, exactly one procedure containing OPTIONS(MAIN). If no main routine exists, the linker complains that your program has no starting address. If more than one main routine exists, the linker complains that there are duplicate references to the name main.

Every .DLL that you build must have at least one module compiled with the DLLINIT compile-time option (see “DLLINIT” on page 56).

Starting the linker

Once the compiler has created object modules out of your source files, use the linker to link them together with the PL/I runtime libraries to create an EXE or DLL file.

Statically linking

To statically link the library into your .EXE, specify the LIBS(SINGLE STATIC) or LIBS(MULTI STATIC) compile-time option (see “LIBS” on page 73). You must also link with the /NOE linker option.

Linking from the command line

Specify the ILINK command followed by any sequence of options, file names, or directories, separated by space or tab characters.

- **options**
  One or more ILINK options. ILINK options start with a / or - character.

- **filename**
  The names of one or more of the following kinds of files:
  - Object files—have an .OBJ filename extension
  - Library files—have an .LIB filename extension
  - Definition files—have a .DEF filename extension
  - Export files—have an .EXP filename extension
  - Resource files—have an .RES filename extension
  You must specify at least one object file to use ILINK correctly.

- **directories**
  One or more directory locations which end with a / or \ character.

- **responsefile**
  The name of a response file. The file name should immediately follow the @ character.

Linking considerations

You can specify the name of the output file with the /OUT option. You can specify the name of a map file with the /MAP option.
In addition to the libraries you specify, by default the linker searches the PL/I runtime libraries defined in your object files at compile time (see “LIBS” on page 73).

The directories you specify become part of the linker's search path, before any directories set in the LIB environment variable. See “Search rules” on page 188 and “Specifying directories” on page 188 for more information.

You can use wildcard characters to specify multiple object files. For example, use *.OBJ to specify all the object files in a directory.

**Filename extensions are not assumed**

The linker does not assume extensions for files. If you specify a filename with no extension, then the linker looks for the file with that name and no extension. If the linker cannot find a file, it stops linking.

**Examples**

The following command links the object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. The linker searches for unresolved external references in the library file XLIB.LIB and in the default libraries. Since there is no name provided for the executable file, it is named FUN.EXE, taking the filename of the first object file and the default extension .EXE. The linker also produces a map file, FUNLIST.MAP.

```
ilink /MAP:funlist fun.obj text.obj table.obj care.obj xlib.lib
```

The following command links the files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ into an executable file named MAIN.EXE, and produces a map file named MAIN.MAP.

```
ilink /MAP main.obj getdata.obj printit.obj
```

In Windows, the same command changes slightly by adding an export file, GETDATA.EXP, which specifies the functions that are exported from GETDATA.DLL.

```
ilink getdata.obj printit.obj /OUT:getdata.dll /DLL getdata.exp
```

**Linking from a make file**

Use a make file to organize the sequence of actions (such as compiling and linking) required to build your project. You can then invoke all the actions in one step. The NMAKE utility saves you time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files.

The following figure contains a basic make file example.
The linker is designed to link object files with other library files you specify to produce either an executable program file (.EXE) or a dynamic link library (.DLL).

The linker optionally produces a map file, which provides information about the contents of the executable output.

**Input Output**

<table>
<thead>
<tr>
<th>Options</th>
<th>executables</th>
<th>object files (*OBJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map file (*.MAP)</td>
<td>library files (*.LIB)</td>
<td>return code</td>
</tr>
<tr>
<td>module definition file (*.DEF)</td>
<td>export files (*.EXP)</td>
<td>resource files (*.RES)</td>
</tr>
</tbody>
</table>
Search rules

When searching for an object (.OBJ), library (.LIB), or module definition (.DEF) file, the linker looks in the following locations in this order:

1. The directory you specified for the file or the current directory if you did not give a path. Default libraries do not include path specifications.

   Note: If you specify a path with the file, the linker searches only that path.

2. Any directories entered by themselves on the command line (they must end with a slash (/) or backslash (\) character). See the section on "Specifying directories" for more information.

3. Any directories listed in the LIB environment variable.

If the linker cannot locate a file, it generates an error message and stops linking.

Example

A response file could contain the following information:

```
FUN.OBJ TEXT.OBJ TABLE.OBJ CARE.OBJ
NEWLIBV3.LIB
C:\TESTLIB\ 
```

The linker links four object files to create an executable file named FUN.EXE. The linker searches NEWLIBV3.LIB before searching the default libraries to resolve references.

To locate NEWLIBV3.LIB and the default libraries, the linker searches the following locations in this order:

1. The current directory (because NEWLIBV3.LIB was entered without a path)
2. The C:\TESTLIB\ directory
3. The directories listed in the LIB environment variable

Specifying directories

To have the linker search additional directories for input files, specify a drive or directory by itself on the command line. Specify the drive or directory with a slash (/) or backslash (\) character at the end so the linker will recognize it as a path.

The linker searches the paths you specify before it searches the paths in the LIB environment variable. See the section on "Search rules" for more information.

Filename defaults

If you do not enter a file name, the linker assumes the following defaults:

<table>
<thead>
<tr>
<th>Table 7. Linker filename defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File</strong></td>
</tr>
<tr>
<td>Object files</td>
</tr>
<tr>
<td>Output file</td>
</tr>
<tr>
<td>Map file</td>
</tr>
<tr>
<td>Library files</td>
</tr>
<tr>
<td>Module definition file</td>
</tr>
</tbody>
</table>
Specifying object files

When you invoke the linker from the command line, the linker assumes that any input it cannot recognize as other files, options, or directories must be an object file. Use a space or tab character to separate files. See “Linking from the command line” on page 185 for more information on how the linker interprets input.

You can also use wildcard characters to specify multiple object files. For example, use *.OBJ to specify all the object files in a directory.

Using response files

Instead of specifying linker input on the command line, you can put options and filename parameters in a response file. You can combine the response file with options and parameters on the command line.

When you invoke the linker, use the following syntax:

```
ilink @responsefile
```

The value for `responsefile` is the name of the response file. The @ symbol indicates that the file is a response file. If the file is not in the working directory, specify the path for the file as well as the file name.

You can begin using a response file at any point on the linker command line. Although multiple response files can be specified on the command line, they cannot be nested.

Options can appear anywhere in the response file. If an option is not valid, the linker generates an error message and stops linking.

Specify the contents of the response file just as you would on the command line. Because the default syntax identifies input by file extension rather than by position on the command line, it does not matter how many lines there are, or whether there are blank lines in the file.

Example

The response file named FUN.LNK contains the following:

```
/DEBUG  /MAP
fun.obj  text.obj  table.obj  care.obj
/exec
/map:funlist
graf.lib
```

When you enter `ilink @fun.lnk`, the linker does the following:

- Links the four object modules `fun.obj`, `text.obj`, `table.obj`, and `care.obj` into an .EXE file named `fun.exe`. Because no output type is specified, the linker defaults to .exe.
- Generates the map file `funlist.map` (assuming the extension .map).
- Preserves debugging information (because of the /DEBUG option).
- Links any needed routines from the library file `graf.lib`, and from the default PL/I libraries specified in the object files.
Specifying executable output type

You can use the linker to produce executable modules (.EXE) and dynamic link libraries (.DLL). The linker produces .EXE files by default.

Use options to specify what kind of output you want:
- To produce a .DLL, specify the /DLL option. Or, include the module statement LIBRARY.

Producing an .EXE file
The linker produces .EXE files by default. Use the /EXEC option, to explicitly identify the output file as an .EXE file.

An .EXE file is one that can be executed directly. You can run the program by typing the name of the file. In contrast, DLL and device driver programs execute when they are called by other processes, and cannot be run independently. To reduce the size of the .EXE file and improve its performance, use the following options:
- /ALIGNFILE:n to set the file alignment for sections in the output file. Set n to smaller factors to reduce the size of the executable, and to larger factors to reduce load time for the executable. By default, the alignment is set to 512.
- /BASE:n to specify the load address for the executable. For example, if several DLLs are loaded at base addresses that ensure that the DLLs do not overlap, the linker does not have to reapply the relocation records. n (the load address) must be a multiple of 0x10000, and it cannot be 0.

If you do not specify an extension for the output file name, the linker automatically adds the extension .EXE to the name you provide. If you do not specify an output filename at all, the linker generates an .EXE file with the same filename as the first .OBJ file it linked.

Producing a dynamic link library
A dynamic link library (.DLL) file contains executable code for common functions, just as a library (.LIB) file does. When you link with a DLL (using an import library), the code in the DLL is not copied into the executable file. Instead, only the import definitions for DLL functions are copied, resulting in a smaller executable. At run time, the dynamic link library is loaded into memory, along with the .EXE file.

To produce a DLL as output, compile at least one object file with the DLLINIT compiler option, and link it with the /DLL linker option. You must include an export definition (.EXP) file that specifies which functions are to be included in the DLL.

You can find more information in Chapter 19, “Building dynamic link libraries,” on page 379.

To reduce the size of the DLL and improve its performance, use the following options:
- /ALIGNFILE:value to set the alignment factor in the output file. Set value to smaller factors to reduce the size of the DLL, and to larger factors to reduce load time for the DLL. By default, the alignment is set to 512.
For DLLs, setting a /BASE value can save load time when the given load address is available. If the load address is not available, the /BASE value is ignored, and there is no load time benefit.

Once you have produced the DLL, you can produce an executable that links to the DLL.

The linker determines which functions your object files need during the linking process. Use the ILIB utility to create an import library, and then use the .LIB file as input to the linker.

### Packing executables

Specify /DBGPACK when you are debugging, to reduce the size of the executable file and potentially improve debugger performance.

### Generating a map file

Specify /MAP to generate a map file, which lists the object modules in your output file; section names, addresses, and sizes; and symbol information. If you do not specify a name for the map file, the map file takes the name of the executable output file, with the extension .MAP. To prevent the map file from being generated, use the default, /NOMAP.

Specify /LINENUMBERS to include source file line numbers and associated addresses in the map file.

### Linker return codes

The linker has the following return codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The link was completed successfully. The linker detected no errors, and issued no warnings.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Warnings</strong> issued. There may be problems with the output file.</td>
</tr>
<tr>
<td>8</td>
<td><strong>Errors</strong> detected. The linking might have completed, but the output file cannot be run successfully.</td>
</tr>
<tr>
<td>12</td>
<td>Both warnings issued and errors detected (see return codes 4 and 8)</td>
</tr>
<tr>
<td>16</td>
<td><strong>Severe errors</strong> detected. Linking ended abnormally, and the output file cannot be run successfully.</td>
</tr>
<tr>
<td>20</td>
<td>Both warnings issued and severe errors detected (see return codes 4 and 16)</td>
</tr>
<tr>
<td>24</td>
<td>Both errors and severe errors issued (see return codes 8 and 16)</td>
</tr>
<tr>
<td>28</td>
<td>The linker issued warnings, detected errors, and detected severe errors (see return codes 4, 8, and 16)</td>
</tr>
</tbody>
</table>

If you invoke the linker through a makefile, you can force NMAKE to ignore warnings by putting -7 before the ILINK command.
Linker return codes
Chapter 7. Setting linker options

Linker options are not case sensitive, so you can specify them in lower-, upper-, or mixed case. You can also substitute a dash (-) for the slash (/) preceding the option. For example, -DEBUG is equivalent to /DEBUG. You can specify options in either a short or long form. For example, /DE, /DEB, and /DEBU are all equivalent to /DEBUG. See “Summary of Windows linker options” on page 194 for the shortest acceptable form for each option. Lower- and uppercase, short and long forms, dashes, and slashes can all be used on one command line, as in:

```
ilink /de -DBGPACK -Map /NOI prog.obj
```

Separate options with a space or tab character. You can specify linker options in the following ways:
- On the command line
- In the ILINK environment variable

Options specified on the command line override the options in the ILINK environment variable.

Some linker options take numeric arguments. You can enter numbers in decimal, octal, or hexadecimal format. See “Specifying numeric arguments” on page 194 for more information.

Setting options on the command line

Linker options specified on the command line override any previously specified in the ILINK environment variable (as described in “Setting options in the ILINK environment variable”).

You can specify options anywhere on the command line. Separate options with a space or tab character.

For example, to link an object file with the /MAP option, enter:
```
ilink /M myprog.obj
```

Setting options in the ILINK environment variable

Store frequently used options in the ILINK environment variable. This method is useful if you find yourself repeating the same command-line options every time you link. You cannot specify file names in the environment variable, only linker options.

The ILINK environment variable can be set either from the command line, in a command (.CMD) file, or in the System Properties. If it is set on the command line or by running a command file, the options will only be in effect for the current session (until you reboot your computer). If it is set in the System Properties, the options are set when you boot your computer, and are in effect every time you use the linker unless you override them using a .CMD file or by specifying options on the command line.

Using the linker

In the following example, options on the command line override options in the environment variable. If you enter the following commands:
The ILINK environment variable

SET ILINK=/NOI /AL:256 /DE
ILINK test
ILINK /NODEF /NODEB prog

The first command sets the environment variable to the options
/NOIGNORECASE, /ALIGNMENT:256, and /DEBUG

The second command links the file test.obj, using the options specified in the
environment variable, to produce test.exe

The last command links the file prog.obj to produce prog.exe, using the option
/NODEFAULTLIBRARYSEARCH, in addition to the options /NOIGNORECASE
and /ALIGNMENT:256. The /NODEBUG option on the command line overrides
the /DEBUG option in the environment variable, and the linker links without the
/DEBUG option.

Specifying numeric arguments

Some linker options and module statements take numeric arguments. You can
specify numbers in any of the following forms:

Decimal
Any number not prefixed with 0 or 0x is a decimal number. For example,
1234 is a decimal number.

Octal
Any number prefixed with 0 (but not 0x) is an octal number. For example,
01234 is an octal number.

Hexadecimal
Any number prefixed with 0x is a hexadecimal number. For example,
0x1234 is a hexadecimal number.

Summary of Windows linker options

Table 8. Windows linker options summary

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;?&quot;</td>
<td>Display help</td>
<td>None</td>
</tr>
<tr>
<td>&quot;ALIGNADDR&quot;</td>
<td>Set address alignment</td>
<td>/A:0x00010000</td>
</tr>
<tr>
<td>&quot;ALIGNFILE&quot;</td>
<td>Set file alignment</td>
<td>/A:512</td>
</tr>
<tr>
<td>&quot;BASE&quot;</td>
<td>Set preferred loading address</td>
<td>/BASE:0x00400000</td>
</tr>
<tr>
<td>&quot;CODE&quot;</td>
<td>Set section attributes for executable</td>
<td>/CODE:RX</td>
</tr>
<tr>
<td>&quot;DATA&quot;</td>
<td>Set section attributes for data</td>
<td>/DATA:RW</td>
</tr>
<tr>
<td>&quot;DBGPACK&quot;, /NODBGPACK&quot;</td>
<td>Pack debugging information</td>
<td>/NODB</td>
</tr>
<tr>
<td>&quot;DEBUG&quot;, /NODEBUG&quot;</td>
<td>Include debugging information</td>
<td>/NODB</td>
</tr>
<tr>
<td>&quot;DEFAULTLIBRARYSEARCH&quot;</td>
<td>Search default libraries</td>
<td>/DEF</td>
</tr>
<tr>
<td>&quot;DLL&quot;</td>
<td>Generate DLL</td>
<td>/EXEC</td>
</tr>
<tr>
<td>&quot;EXECUTABLE&quot;</td>
<td>Specify an entry point in an executable file</td>
<td>None</td>
</tr>
<tr>
<td>&quot;EXTDICTIONARY&quot;</td>
<td>Generate .EXE file</td>
<td>/EXEC</td>
</tr>
<tr>
<td>/NOEXTDICTIONARY&quot;</td>
<td>Use extended dictionary to search libraries</td>
<td>/EXT</td>
</tr>
</tbody>
</table>
Table 8. Windows linker options summary (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>/EXTDICTIONARY</td>
<td>Do not relocate the file in memory</td>
<td>/NOEXITDICTIONARY</td>
</tr>
<tr>
<td>/NOEXITDICTIONARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/FORCE</td>
<td>Create executable output file even if errors</td>
<td>/NOFORCE</td>
</tr>
<tr>
<td></td>
<td>are detected</td>
<td></td>
</tr>
<tr>
<td>/HEAP</td>
<td>Set the size of the program heap</td>
<td>/HEAP:0x100000,0x1000</td>
</tr>
<tr>
<td>/HELP</td>
<td>Display help</td>
<td>None</td>
</tr>
<tr>
<td>/INCLUDE</td>
<td>Forces a reference to a symbol</td>
<td>None</td>
</tr>
<tr>
<td>/INFORMATION, /NOINFORMATION</td>
<td>Display status of linking process</td>
<td>/NOIN</td>
</tr>
<tr>
<td>/LINENUMBERS, /NOLINENUMBERS</td>
<td>Include line numbers in map file</td>
<td>/NOLI</td>
</tr>
<tr>
<td>/LOGO, /NOLOGO</td>
<td>Display logo, echo response file</td>
<td>/LO</td>
</tr>
<tr>
<td>/MAP, /NOMAP</td>
<td>Generate map file</td>
<td>/NOM</td>
</tr>
<tr>
<td>/OUT</td>
<td>Name output file</td>
<td>Name of first .obj file</td>
</tr>
<tr>
<td>/PMTYPE</td>
<td>Specify application type</td>
<td>/PMTYPE:V10</td>
</tr>
<tr>
<td>/SECTION</td>
<td>Set attributes for section</td>
<td>Set by /CODE and /DATA</td>
</tr>
<tr>
<td>/SEGMENTS</td>
<td>Set maximum number of segments</td>
<td>/SE:256</td>
</tr>
<tr>
<td>/STACK</td>
<td>Set stack size of application</td>
<td>/STACK:0x100000,0x1000</td>
</tr>
<tr>
<td>/STUB</td>
<td>Specify the name of the DOS stub file</td>
<td>None</td>
</tr>
<tr>
<td>/SUBSYSTEM</td>
<td>Specify the required subsystem and version</td>
<td>/SUBSYSTEM:WINDOWS,4.0</td>
</tr>
<tr>
<td>/VERBOSE</td>
<td>Display status of linking process</td>
<td>/NOV</td>
</tr>
<tr>
<td>/VERSION</td>
<td>Write a version number in the run file</td>
<td>/VERSION:0.0</td>
</tr>
</tbody>
</table>

Windows linker options

This section describes the linker options in alphabetical order.

For each option, the description includes:

- The syntax for specifying the option.
- The default setting.
- Any accepted abbreviations.
- A description of the option and its parameters, and any interaction it may have with other options.

/?

Use /? to display a list of valid linker options. This option is equivalent to /HELP.

/ALIGNADDR

Use /ALIGNADDR to set the address alignment for segments.
Windows linker options

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 256M.

Default: /ALIGNADDR:0x00010000

Abbreviation: /ALIGN

/ALIGNFILE

Use /ALIGNFILE to set the file alignment for segments.

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 64K.

Default: /ALIGNFILE:512

Abbreviation: /A

(BASE)

Use /BASE to specify the preferred load address for the first load segment of a .DLL file.

Specifying @filename, key, in place of address, bases a set of programs (usually a set of DLLs) so they do not overlap in memory. filename is the name of a text file that defines the memory map for a set of files. key is a reference to a line in filename beginning with the specified key. Each line in the memory-map file has the syntax: key address maxsize

Separate the elements with one or more spaces or tabs. The key is a unique name in the file. The address is the location of the memory image in the virtual address space. The maxsize is an amount of memory within which the image must fit. The linker will issue a warning when the memory image of the program exceeds the specified size. A comment in the memory-map file begins with a semicolon (;) and runs to the end of the line.

Default: /BASE:0x00400000

Abbreviations: /BAS

.CODE

Use /CODE to specify the default attributes for all code sections. Letters can be specified in any order.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>E or X</td>
<td>EXECUTE</td>
</tr>
<tr>
<td>R</td>
<td>READ</td>
</tr>
<tr>
<td>S</td>
<td>SHARED</td>
</tr>
<tr>
<td>W</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

Default: /CODE:RX
Windows linker options

CODE description abbreviations: None

/DATA

Use /DATA to specify the default attributes for all data sections. Letters can be specified in any order.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>E or X</td>
<td>EXECUTE</td>
</tr>
<tr>
<td>R</td>
<td>READ</td>
</tr>
<tr>
<td>S</td>
<td>SHARED</td>
</tr>
<tr>
<td>W</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

Default: /DATA:RW

Abbreviations: None

/DBGPACK, /NODBGPACK

Use /DBGPACK to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.

Performance Consideration: Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can actually speed up your linking, because there is less information to write to file.

When you specify /DBGPACK, /DEBUG is turned on by default.

Default: /NODBGPACK

Abbreviations: /DB|/NODB

/DEBUG, /NODEBUG

Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with Performance Analyzer. The linker will embed symbolic data and line number information in the output file.

For debugging, compile the object files with TEST.

For the Performance Analyzer, compile the object files with PROFILE and GONUMBER. Linking with /DEBUG increases the size of the executable output file.

Default: /NODEBUG

Abbreviations: /D|/NODEB

/DEFAULTLIBRARYSEARCH

Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references.
Windows linker options

If you specify a library with the option, the linker adds the library name to the list of default libraries. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

Use `/NODEFAULTLIBRARYSEARCH` to tell the linker to ignore default libraries when it resolves external references. If you specify a library with the option, the linker ignores that default library, but searches the rest of the default libraries (and any others that are defined in the object files).

If you specify `/NODEFAULTLIBRARYSEARCH` without specifying library, then you must explicitly specify all the libraries you want to use, including VA PL/I runtime libraries.

Default: `/DEFAULTLIBRARYSEARCH`

Abbreviations: `/DEF` `/NOD`

/DLL

Use `/DLL` to identify the output file as a dynamic link library (.DLL file). The object files should be compiled with the PL/I option DLLINIT.

If you specify `/DLL` with `/EXEC`, only the last specified of the options takes effect.

If you do not specify `/DLL`, or any of the other options, then by default the linker produces an .EXE file (`/EXEC`).

Default: `/EXECUTABLE`

Abbreviation: `/EXEC`

/ENTRY

Use `/ENTRY` to specify an entry point (name of a routine or function) in an executable.

Default: None

Abbreviation: `/EN`

/EXECUTABLE

Use `/EXEC` to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default.

If you specify `/EXEC` with `/DLL`, only the last specified of the options takes effect.

If you do not specify `/EXEC` or `/DLL`, then by default the linker produces an .EXE file.

Default: `/EXECUTABLE`

Abbreviation: `/EXEC`
/EXTDICTIONARY, /NOEXTDICTIONARY

Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see if that module requires other modules in the library, and then pulls in the additional modules automatically.

The linker searches the extended dictionary by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are defining a symbol in your object code that is also defined in one of the libraries to which you are linking. Otherwise the linker issues an error because you have defined the same symbol in two different places. When you link with /NOEXTDICTIONARY, the linker searches the dictionary directly, instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

Default: /EXTDICTIONARY

Abbreviations: /EXT|/NOE

/FIXED, /NOFIXED

Use /FIXED to tell the loader not to relocate a file in memory when the specified base address is not available.

For more information on base addresses, see the /BASE linker option.

Default: /NOFIXED

Abbreviations: /FI|/NOFI

/FORCE

Use /FORCE to produce an executable output file even if there are errors during the linking process.

Default: /NOFORCE

Abbreviations: /FO|/NOFO

/HEAP

Use /HEAP to set the size of the program heap in bytes. The reserve argument sets the total virtual address space reserved. The commit sets the amount of physical memory to allocate initially. When commit is less than reserve, memory demands are reduced, but execution time can be slower.

Default: /HEAP:0x100000,0x1000

Abbreviation: /HEA

/HELP

Use /HELP to display a list of valid linker options. This option is equivalent to /?.

Default: None
Windows linker options

/INCLUDE

Use /INCLUDE to force a reference to a symbol. The linker searches for an object module that defines the symbol.

Default: None

Abbreviation: /INC

/INFORMATION, /NOINFORMATION

See the description of the /VERBOSE linker option.

Default: /NOINFORMATION

Abbreviations: /I /NOIN

/LINENUMBERS, /NOLINENUMBERS

Use /LINENUMBERS to include source file line numbers and associated addresses in the map file. For this option to take effect, there must already be line number information in the object files you are linking.

When you compile, use the GONUMBER option to include line numbers in the object file (or the TEST option to include all debugging information).

If you give the linker an object file without line number information, the /LINENUMBERS option has no effect.

The /LINENUMBERS option forces the linker to create a map file, even if you specified /NOMAP.

By default, the map file is given the same name as the output file, plus the extension .map. You can override the default name by specifying a map filename.

Default: /NOLINENUMBERS

Abbreviations: /L /NOLI

/LOGO, /NOLOGO

Use /NOLOGO to suppress the product information that appears when the linker starts.

Specify /NOLOGO before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

Default: /LOGO

Abbreviations: /LO /NOL
Windows linker options

/\MAP, /\NOMAP
Use /MAP to generate a map file called name. The file lists the composition of each segment, and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name and in order of address.

If you do not specify a directory, the map file is generated into the current working directory. If you do not specify name, the map file has the same name as the executable output file, with the extension .map.

Default: /NOMAP
Abbreviations: /M /NOM

/\OUT
Use /\OUT to specify a name for the executable output file.

If you do not provide an extension with name, then the linker provides an extension based on the type of file you are producing:
File produced
   Default extension
Executable program
   .EXE
Dynamic link library
   .DLL

If you do not use the /\OUT option, then the linker uses the filename of the first object file you specified, with the appropriate extension.

Default: Name of first .OBJ file with appropriate extension.
Abbreviation: /O

/\PMTYPE
Use /\PMTYPE to specify the type of .EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs).

One of the following types must be specified:
   PM  The executable must be run in a window.
   VIO The executable can be run either in a window or in a full screen.
   NOVIO The executable must not be run in a window; it must use a full screen.

Default: /PMTYPE:VIO
Abbreviation: /PM

/\SECTION
Use /\SECTION to specify memory-protection attributes for the name section. name is case sensitive. You can specify the following attributes:
Letter Sets Attribute
E or X EXECUTE
R  READ
Windows linker options

S       SHARED
W       WRITE

The following example sets the READ and SHARED attributes, but not the EXECUTE, or WRITE attributes, for the section dseg1 in an .EXE file.
/SEC:dseg1,RS

Defaults

Sections are assigned attributes by default, as follows:

Segment
  Default Attributes

Code sections
  EXECUTE, READ (ER)

Data sections (in .EXE file)
  READ, WRITE (RW), not shared

Data sections (in .DLL file)
  READ, WRITE, not shared

CONST32_RO section
  READ, SHARED (RS)

Default: Depends on segment type

Abbreviation: /SEC

/SEGMENTS

Use /SEGMENTS to set the number of logical segments a program can have. You can set number to any value in the range 1 to 16375. See “Specifying numeric arguments” on page 194.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting number to the actual number of segments in the program.

Default: /SEGMENTS:256

Abbreviation: /SE

/STACK

Use /STACK to set the stack size (in bytes) of your program. The size must be an even number from 0 to 0xFFFFFFFF. If you specify an odd number, it is rounded up to the next even number.
Windows linker options

`reserve` indicates the total virtual address space reserved. `commit` sets the amount of physical memory to allocate initially. When `commit` is less than `reserve`, memory demands are reduced, although execution time may be slower.

Default: `/STACK:0x100000,0x1000`

Abbreviation: `/ST`

/STUB

Use `/STUB` to specify the name of the DOS executable at the beginning of the output file created.

Default: None

Abbreviation: `/STU`

/SUBSYSTEM

Use `/SUBSYSTEM` to specify the subsystem and version required to run the program. The `major` and `minor` arguments are optional and specify the minimum required version of the subsystem. The `major` and `minor` arguments are integers in the range 0 to 65535.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Major.Minor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOWS</td>
<td>3.10</td>
<td>A graphical application that uses the Graphical Device Interface (GDI) API.</td>
</tr>
<tr>
<td>CONSOLE</td>
<td>3.10</td>
<td>A character-mode application that uses the Console API.</td>
</tr>
</tbody>
</table>

Default: `/SUBSYSTEM:WINDOWS,4.0`

Abbreviation: `/SU`

/VERBOSE

Use `/VERBOSE` to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

If you are having trouble linking because the linker is finding the wrong files or finding them in the wrong order, use `/VERBOSE` to determine the locations of the object files being linked and the order in which they are linked.

The output from this option is sent to `stdout`. You can redirect the output to a file using Windows redirection symbols.

`/VERBOSE` is the same as `/INFORMATION`.

Default: `/NOVERBOSE`

Abbreviations: `/VERB`, `/NOV`

/VERSION

Use `/VERSION` to write a version number in the header of the run file. The `major` and `minor` arguments are integers in the range 0 to 65535.
Windows linker options

Default: /VERSION:0.0

Abbreviation: /VER
Part 3. Running and debugging your program

Chapter 8. Using run-time options
---
Chapter 9. Testing and debugging your programs
Chapter 8. Using run-time options

Once you have prepared the executable form of your PL/I program, you need to test its execution behavior. The first step is to run the program and see what happens. Depending on the nature of your application, you might need to do some input and output setup (SET statements) before invoking the program.

Setting run-time environment variables

You can set the run-time environment for your program by using environment variables.

**PATH**

Use the PATH environment variable to specify the search path for EXE and CMD files not in the current directory.

```
set path=c:\ibm;d:\project
```

You can specify one or more directories with this variable. Given the preceding example, the current directory is searched first, followed by c:\ibm and then d:\project.

**DPATH**

Use DPATH to specify the search path for run-time messages. The program searches for them first in the current directory, then in the directory or directories specified by the DPATH variable. The following example would cause the program to search the current directory followed by c:\set1 and d:\set2 in that order.

```
set dpath=c:\set1;d:\set2
```

Specifying run-time options

Each time your application executes, a set of run-time options is established. These options determine some of the properties of the application’s execution, such as allocation of storage and production of reports. IBM supplies defaults for each of the run-time options; however, you can change them as needed prior to running your application.

Where to specify run-time options

You can alter the default settings for run-time options in an environment variable and in the application source code. Alternatives, from lowest priority to highest priority, are:

- **Using the IBM defaults**
- **Setting run-time options in the CEE.OPTIONS environment variable**

Use the SET command on the command line or define them in System Properties to specify run-time options by means of the CEE.OPTIONS environment variable. For example:

```
set cee.options=natlang(enu)
```

As mentioned above, there are two methods for setting options in the CEE.OPTIONS environment variable. The first method, setting CEE.OPTIONS in System Properties has lower priority than the second, using the SET command.
Specifying run-time options

1. Setting CEE.OPTIONS in System Properties
   Run-time options specified in System Properties are the options in effect for every session you start. This is a good place to specify run-time options that you want to have in effect for every application you run.
   If CEE.OPTIONS already exists in the System Properties, change or add to the existing variable.

2. Run-time options specified in a SET command on the command line are in effect only for that session or window and override any run-time options specified in System Properties. This is the recommended method.
   To change run-time option settings, use a SET command with the desired settings. Each SET command completely replaces any previous SET commands, including the definition in System Properties. Therefore, you must include the settings of all run-time options from any previous SET command if you still want them in effect in each subsequent SET command.
   For example, assume you have the following in your System Properties:
   ```
   cee.options=natlang(jpm)
   ```
   and later enter this command from the command line:
   ```
   set cee.options=natlang(enu)
   ```
   This means that NATLANG has returned to its default value, which is NATLANG(ENU).
   To return all run-time options to the IBM-supplied defaults, set CEE.OPTIONS to a null argument:
   ```
   set cee.options=
   ```
   With Windows, you can group several commands, including a SET command for CEE.OPTIONS, in a command file. Running such a command file is equivalent to issuing each of the commands individually on the command line.

Specifying multiple run-time options or suboptions

When specifying a string of run-time options, you must separate each option with a comma without any embedded spaces.

Use commas to separate suboptions of run-time options. If you do not specify a suboption, you must still specify the comma to indicate its omission. Trailing commas are not required. If you do not specify any suboptions, the defaults are used. For example, NATLANG() is valid syntax.

Default settings for the options are indicated in the options syntax diagrams or in the descriptions of suboptions, where applicable.

Run-time options

This section describes the run-time option NATLANG.

NATLANG

The NATLANG option specifies the national language to be used for run-time messages. Message translations are provided for Japanese and mixed-case U.S. English. NATLANG also determines how the message facility formats messages.
Run-time options

JPN
This is a 3-character id specifying Japanese. Message text can be a mixture of SBCS (single-byte character set) and DBCS (double-byte character set) characters.

ENU
This is a 3-character id specifying mixed-case U.S. English. Message text is made up of SBCS characters and consists of both upper and lowercase letters.

USAGE: NATLANG(ENU)

Run-time option and storage reports, as well as dump output, are written only in mixed-case U.S. English.

If you specify a national language that is unavailable on your system, the default is used.

Shipping run-time DLLs

If you are shipping DLLs with your application, this list should help you determine which ones you need based on your application.

For Windows, the following files are needed for a non-multithreading application:
- BIN\HEPWS20.DLL
- BIN\IBMWS20.DLL
- BIN\IBMWSTB.DLL
- BIN\IBMWS20F.DLL
- BIN\IBMWS20G.DLL
- BIN\IBMRTENU.DLL

These files are needed for multithreading applications on Windows:
- BIN\HEPWM20.DLL
- BIN\IBMWM20.DLL
- BIN\IBMWMTB.DLL
- BIN\IBMWM20F.DLL
- BIN\IBMWM20G.DLL
- BIN\IBMRTENU.DLL

In addition to those listed previously, if your Windows application uses BTRIEVE, you must also ship BIN\IBMPBTRV.DLL.

Your application containing a copy of any of these files or modules must be labeled as follows:

CONTAINS
IBM PL/I for Windows Version 8.5
Runtime Modules
(c) Copyright IBM Corporation 2012
All Rights Reserved
Shipping run-time DLLs
Chapter 9. Testing and debugging your programs

Effective design and coding practices help you create quality programs and should be followed by thorough testing of those programs. You should give adequate attention to the testing phase of development so that:

- Your program becomes fully operational after the fewest possible test runs, thereby minimizing the time and cost of program development.
- Your program is proven to have fulfilled all of its design objectives before it is released for production work.
- Your program contains sufficient comments to enable those who use and maintain the program to do so without additional assistance.

The process of testing usually uncovers bugs, a generic term that encompasses anything that your program does that you did not expect it to do. The process of removing these bugs from your program is called debugging.

While this chapter does not attempt to provide an exhaustive coverage of testing and debugging, it does provide useful tips and techniques to help you produce top-quality, error-free PL/I programs. Both general and PL/I-specific testing and debugging information follow.

Testing your programs

Testing your PL/I programs can be difficult, especially if the programs are logically complex or involve numerous modules. Do not skip this step, though, because it is important to detect and remove bugs from a program before it moves into a production environment.

Here are three testing approaches that you can apply to all of your PL/I programs:

**Code inspection**

Also called desk checking, code inspection involves selecting a piece of code and reading it from the viewpoint of the computer. With either a printed copy of the source program or an online view of the source file, follow the flow of the program. Where there is input data, guess at some likely data and substitute it for variable values. When there is a calculation, do the calculation manually or with a calculator, and so on. Code inspection often reveals logic problems, syntax errors, and bugs that the compiler misses (for example, “n + 2” instead of “n*2”).

**Data testing**

You provide a program with test data to verify that it runs as designed. The purpose of data testing is to see if the program takes exception (for example, a run-time error) to any possible data that it might have to handle in a production environment. Therefore, you need to use a wide variety of data to test your program.

For example, have your program process extremes of data that you know lead to errors (such as the OVERFLOW condition) and see how the program responds. Your program should incorporate error checking (such as ERROR ON-units) to accommodate any possible data.
Testing your programs

**Attention:** You should never test with irreplaceable data, nor should you store irreplaceable data within access of a program being tested!

**Path testing**

The data that you use for testing a program should be selected to test all parts of the program. In other words, if your program consists of a number of modules, the data that you test the program with should require the use of all of the modules. If your program can take five possible paths at a given point, you should provide sets of data that take the program down each of the five paths.

As your program becomes more and more complex, providing the program with data to accommodate every possible path combination might become practically impossible. However, it is important that you select test cases that check a representative range of paths. For example, rather than check every possible iteration of a DO-loop, test the first, last, and one intermediate case.

Bugs are discovered as you test your programs and removing those bugs sometimes requires being able to reproduce them. Therefore, when you test programs, always begin from a known state. For example, when a bug is encountered you should know the values of variables, the compile-time options used, the contents of memory, and so on. PL/I provides features such as SNAP and PLIDUMP that help you do this.

As a rule, a program that ran perfectly well yesterday but reveals a bug today does so because of one or more changes to the state of the machine. Therefore, when testing your PL/I programs be sure to know, in detail, the state of the machine at compile time and at run time.

**General debugging tips**

Debugging is a process of letting your program run until it does something that you did not expect it to do. After finding a bug, you modify the program so that it does not encounter the bug when the program is in the exact machine state that initially produced the bug. This is accomplished by a combination of back-tracking, intuition, and trial and error. The major obstacle to effective debugging is that removing one bug can introduce new bugs into your program. You should consider general debugging tips as well as some debugging techniques specific to PL/I.

Consider the following tips when debugging your programs:

**Make one change at a time**

When attempting to remedy a bug, introduce only one change into the source code of your program at a time. By introducing a single change, you can compare the program behavior before and after the change to accurately measure the effect of the change.

**Follow program logic sequence**

Fix your program’s bugs in the order in which they are encountered when the program is run.

**Watch for unexpected results**

Locate a given bug in the program source code at a point that corresponds to an unexpected change in the state of program execution.

For example, the undesired change in the state of program execution might be the unintended assignment of the decimal value “100” to the character variable
“z”. In this case, you might find that the source code has an error that assigns the wrong variable in an assignment statement.

## PL/I debugging techniques

PL/I provides you with a number of methods for program debugging which are described in the following sections:

- Compile-time options
- Footprints
- Dumps
- Error and condition handling

### Using compile-time options for debugging

The PL/I workstation products are designed to diagnose many of the bugs in your programs at compile time, and provides you with a compiler listing that explains what mistakes you made and where you made them. In addition, you can use compile-time options to make the compiler listing even more useful.

The following compile-time options are useful for debugging your PL/I programs:

- **FLAG**
  
  Suppresses the listing of diagnostic messages below a certain severity and terminates compilation if a specified number of messages is reached. If your program is not behaving as expected and the compiler messages do not explain the problem, you might want to use FLAG to include informational messages in the compiler listing. These messages (otherwise suppressed by default) might help explain problems in your program. For additional information on using FLAG, see “FLAG” on page 59.

- **GONUMBER**
  
  Creates a statement number table that is needed for debugging.

- **PREFIX**
  
  Enables or disables specified PL/I conditions. Because you can specify the conditions with a compile-time option, you do not need to change your source program. Compiling with PREFIX(SUBRG STRZ STRG) can be very helpful in debugging. For more information on using PREFIX, see “PREFIX” on page 110.

- **RULES**
  
  Specifies the strictness with which various language rules are enforced by the compiler. You can use it to flag common programming errors.

  You might find the following suboptions for RULES particularly useful for debugging:

  - **NOLAXIF**
    
    Disallows IF, WHILE, UNTIL, and WHEN clauses to evaluate to other than BIT(1) NONVARYING.

  - **NOLAXDCL**
    
    Disallows all implicit and contextual declarations except for built-ins and the files SYSIN and SYSPRINT.

  - **NOLAXQUAL**
    
    The compiler flags any reference to structure members that are not level 1 and are not dot qualified.

For example, consider the program:
PL/I debugging techniques

```plaintext
program: proc( ax1xcb, ak2xcb );
dcl (ax1xcb, ax2xcb ) pointer;
dcl
  1 xcb based,
  2 xcba13 fixed bin,...
ax1xcb->xcba13 = ax2xcb->xcba13;
```

With RULES(NOLAXDCL) in effect, the two typographical errors above are considered implicit declarations by the compiler and are flagged as errors. For more information on using RULES, see "RULES" on page 118.

**SNAP**

Specifies that the compiler produces a listing of trace information that is useful for locating errors in your program.

For detailed information on using SNAP for debugging, see "SNAP dumps for trace information" on page 219.

For more information on SNAP syntax, see "SNAP" on page 127.

**XREF**

Specifies that the compiler listing includes a table of names used in the program together with the numbers of the statements in which they are referenced or set. This allows you to easily track where names are used in your source program. For more information on using XREF, see "XREF" on page 145.

**Using footprints for debugging**

When debugging, it is useful to periodically check:

- Where your program is in its execution flow (for example, which module is being run).
- The value of identifiers so that you can see when they change and what values they are assigned.

To accomplish these tasks, you can use built-in functions, PUT DATA and PUT LIST statements, and display statements. These approaches are described in more detail in the following sections.

**Built-in functions**

The built-in functions PROCNAME, PACKAGENAME, and SOURCELINE are useful in following the execution of your program when you are trying to track the location of a problem and the sequence of events that caused it. The following statement can be inserted wherever you want to display the procedure name and line number of the statement currently being executed.

```plaintext
display (procname() || sourceline());
```

**PUT LIST**

Allows you to transmit strings and data items to the data stream (for example, to a printer-destined output file). For example, the following procedure lets you know if the FIXEDOVERFLOW condition is raised, and prints out the value of the variable that led to the condition (in this case, z):

```plaintext
Debug: Proc(x);
dcl x fixed bin(31);
on fixedoverflow
  begin;
    put skip list('Fixedoverflow raised because z = ' || z);
  end;
end;
get list(z);
x = 8 * z;
```
If $z$ is too large, multiplying it by 8 produces a value that is too large for any FIXED BIN(31) variable and would therefore raise the FIXEDOVERFLOW condition. PUT SKIP LIST transmits the data (in this case, the string “Fixedoverflow raised because $z = ...”) to the default file SYSPRINT. You can define SYSPRINT using export DD= statements. For more information on using SYSPRINT, see “Using SYSIN and SYSPRINT files” on page 265.

**PUT DATA**

Allows you to transmit the value of data items to the output stream. For example, if you specified the following line in your program, it would transmit the values of `string1` and `string2` to the output stream (for example, to SYSPRINT):

```pl
    put data (string1, string2);
```

**DISPLAY**

You can use DISPLAY to transmit information to your monitor. This can be useful to let you know how far a program has progressed, what procedure a program is running, and so on. For example:

```pl
    display ('End of job!');
    display ('Reached the MATH procedure');
    display ('Hurrah! Got past the string manipulation stuff...');
```

Using DISPLAY with PUT statements results in output appearing in unpredictable order. For more information on using the DISPLAY statement, see “DISPLAY statement input and output” on page 253.

### Using dumps for debugging

When you are debugging your programs, it is often useful to obtain a printout (a dump) of all or part of the storage used by your program. You can also use a dump to provide trace information. Trace information helps you locate the sources of errors in your program.

Two types of dumps are useful:

- **PLIDUMP**
- **SNAP**

Use of the IMPRECISE compile-time option might lead to incomplete trace information. For additional information on the IMPRECISE option, see “IMPRECISE” on page 64.

#### Formatted PL/I dumps—PLIDUMP

You use PLIDUMP to obtain:

- Trace information that allows you to locate the point-of-origin of a condition in your source program.
- File information, including: the attributes of the files open at the time of the dump, the values of certain file-handling built-in functions, and the contents of the I/O storage buffer.

To get a formatted PL/I dump, you must include a call to PLIDUMP in your program. The statement CALL PLIDUMP can appear wherever a CALL statement appears. It has the following form:

```pl
    call plidump('dump options string', 'dump title string');
```

**dump options string**

An expression specifying a string consisting of any of the following dump option characters:
PL/I debugging techniques

T–Trace
   PL/I generates a calling trace.

NT–No trace
   The dump does not give a calling trace.

F–File information
   The dump gives a complete set of attributes for all open files, plus the
   contents of all accessible I/O buffers.

NF–No file information
   The dump does not give file information.

S–Stop
   The program ends after the dump.

E–Exit
   The current thread or the program (if it is the main thread) ends after the
   dump.

K    Ignored.

NK   Ignored.

C–Continue
   The program continues after the dump.

PL/I reads options from left to right. It ignores invalid options and, if
contradictory options exist, takes the rightmost options.

dump title string
   An expression that is converted to character if necessary and printed as a
   header on the dump. The string has no practical length limit. PL/I prints this
   string as a header to the dump. If the character string is omitted, PL/I does
   not print a header.
   
   If the program calls PLIDUMP a number of times, the program should use a
different user-identifier character string on each occasion. This simplifies
identifying the point at which each dump occurs. In addition to this header,
each new invocation of PLIDUMP prints another heading above the
user-identifier showing the date, time, and page number 1.

PLIDUMP defaults:

The default dump options are T, F, and C with a null dump title string:
   plidump('TFC', ' ');

Suggested PLIDUMP coding:

A program can call PLIDUMP from anywhere in the program, but the normal
method of debugging is to call PLIDUMP from an ON-unit. Because continuation
after the dump is optional, the program can use PLIDUMP to get a series of
dumps while the program is running.

You can use the DD:plidump environment variable to specify where the PLIDUMP
output should be located, for example:
   set dd:plidump = d:\mydump;

In your PLIDUMP specification, you cannot override other options such
RECSIZE. The default device association for the file is stderr.
PL/I debugging techniques

PLIDUMP example:

When you run the program shown in Figure 5, a formatted dump is produced as shown in Figure 6 on page 218.

TestDump: proc options(main);
   declare
      Sysin input file,
      Sysprint stream print file;
   open file(Sysprint);
   open file(Sysin);
   put skip list('AbCdEfGhIjKlMnOpQrStUvWxYz');
   call IssueDump;

   IssueDump: proc;
      call plidump( ' ', 'Testing PLIDUMP');
   end IssueDump;
end TestDump;

Figure 5. PL/I code that produces a formatted dump

The call to PLIDUMP in the IssueDump procedure does not specify any PLIDUMP options (they appear as the first of the two character strings), so the defaults are used. Also note that the PL/I default files SYSIN and SYSPRINT have been explicitly opened so that the formatted dump displays the contents of their portions of the I/O buffer.
PL/I debugging techniques

1. Time and date when PLIDUMP is called. Each separate PLIDUMP call has this information.

2. Character string specified in the PLIDUMP call (the second of the two strings provided to PLIDUMP) that is useful in helping to identify the dump if a number of dumps are produced.

3. Trace information, delineated by **Calling trace** and **End of calling trace**. This information allows you to trace back through the procedures from which PLIDUMP was called. In the example above, PLIDUMP was called from the procedure ISSUEDUMP which is nested in the TESTDUMP procedure. The hexadecimal offsets of each procedure are also provided in the trace information.

   The trace information is provided by default as the T option and can be suppressed by specifying the NT option for PLIDUMP.

4. File attributes of SYSIN (opened explicitly in the program).

5. ENVIRONMENT options for the file SYSIN.

6. Values of relevant I/O built-in functions for the file SYSIN.

---

**Figure 6. Example of PLIDUMP output**

---
PL/I debugging techniques

7 Contents of the I/O buffer for the SYSIN file. The first column is the hexadecimal address, the following columns are the hexadecimal contents of memory.

8 Contents of the I/O buffer for SYSPRINT. Notice that the second character string supplied to PLIDUMP (AbCd...) is contained in the I/O buffer, as seen by the text representation of the I/O buffer at the right-hand side of the row.

SNAP dumps for trace information
While not a “dump” in the strictest sense, the SNAP compile-time option is used to find out what error conditions are raised in your program and where they are raised. SNAP provides the same trace information provided by PLIDUMP “T” option (see “Formatted PL/I dumps—PLIDUMP” on page 215). Like PLIDUMP, SNAP can be issued multiple times throughout one run of a program.

An example of a call for a SNAP dump is:

```
on attention snap;
```

This statement calls for a SNAP dump if the ATTENTION condition is raised.

Using error and condition handling for debugging
PL/I condition handling is a powerful tool for debugging programs. All errors detected at run-time are associated with conditions. You can handle these conditions in one of the following ways:
- Writing ON-units that specify what your program should do if a given condition is raised
- Accepting the standard system action

Error and condition handling terminology
You should be familiar with several terms used in discussions of PL/I error and condition handling. The terms are listed below:

Established
An ON-unit becomes established when the ON statement is executed. It ceases to be established when an ON or REVERT statement referring to the same condition is executed, or when the associated block is terminated.

Enabled
A condition is enabled when the occurrence of the condition results in the execution of an ON-unit or standard action.

Interrupts and PL/I conditions
Certain PL/I conditions are detected by machine interrupts. Others have to be detected by special testing code either in the run-time library modules or in the compiled program.

Statically and dynamically descendant
Static and dynamic descendant are terms used to define the scope of error-handling features. ON-units are dynamically descendant; that is, they are inherited from the calling procedure in all circumstances. Condition enablement is statically descendant; that is, it is inherited from the containing block in the source program. Statically descendant procedures can be determined during compilation. Dynamically descendant procedures might not be known until run-time. Figure 7 on page 220 shows an example of statically and dynamically descendant procedures.
PL/I debugging techniques

```
(SIZE):B:PROC;
ON ERROR SNAP;
.
.
CALL C;

C:PROC;
END C;

CALL D;
END B;

D:PROC;  
.
.
CALL E;
END D;

END E;
```

Figure 7. Static and dynamic descendant procedures

Statically descendant:
The enablement prefix SIZE in procedure B is inherited only by the contained procedure C, regardless of which procedure calls which.

Dynamically descendant:
The ON-unit ON ERROR SNAP is inherited by any procedure called by B and any subsequently called procedures. Thus, if B calls D, which calls E, the ON-unit is established in procedure E.

Normal return
A normal return is a return from a called block after reaching the END or RETURN statement, rather than reaching a GOTO statement out of a block. In an error-handling context, normal return is taken to mean normal return from the ON-unit. The action taken after normal return from an ON-unit is specified in the PL/I Language Reference.

Standard system action
Standard system action refers to the default PL/I-defined action taken when a condition occurs for which there is no established ON-unit.

Error handling concepts
You should be familiar with the following error handling concepts when you attempt to debug your PL/I programs. For details on condition handling, see the PL/I Language Reference.

System facilities
The operating system offers error-handling facilities. Various situations can cause a machine interrupt, resulting in an entry to the system supervisor. The PL/I control program can use specified routines to define the action that is taken after any of these interrupts. Alternatively, the PL/I control program passes control to ON-units specified by the PL/I programmer.

Language facilities
The PL/I language and its execution environment extend the error-handling facilities offered by the operating system. Numerous situations can cause interrupts
for PL/I, and some situations (such as ENDFILE) can be used to control normal program flow rather than to handle errors. ON-units allow you to obtain control after most interrupts.

If you do not write ON-units to obtain control after interrupts, you can:
- Accept standard system action
- Choose whether certain conditions cause interrupts or not by enabling or disabling those conditions. If the condition is disabled, neither ON-unit nor standard system action is taken when the condition occurs.

The majority of PL/I conditions occur because of errors in program logic or the data supplied. Some, however, are not connected with errors. These are conditions such as ENDFILE, which are difficult to anticipate because they can occur at any time during program execution.

PL/I has both system messages and snap messages:

**System messages**
If an ON-unit contains both SNAP and SYSTEM, the resulting PL/I message is essentially the PL/I SYSTEM message followed by any (or a combination) of the following three lines:
- From offset xxx in a BEGIN block
- From offset xxx in procedure xxx
- From offset xxx in a condition_name ON-unit

These messages are repeated as often as necessary to trace back to the main procedure.

**SNAP messages**
If an ON-unit contains only SNAP, the resulting PL/I message begins as follows. The messages then continue as for SNAP SYSTEM messages.

Condition_name condition was raised at offset xxx in procedure xxx.

**Determining statement numbers from offsets:**
If you want to translate offset numbers into statement numbers, use the following steps:
- Use the OFFSET compile-time option during compilation
- Open the resulting object (.cod) listing file
- Search for and locate the offset in the first column and find the statement number from the last source statement included in the listing.

**Built-ins for condition handling:**

PL/I also provides condition-handling built-in functions and pseudovariables. These allow you to inspect various fields associated with the interrupt and, in certain cases, to correct the contents of fields causing the error.

These built-in functions include:

<table>
<thead>
<tr>
<th>Datafield</th>
<th>ONCOUNT</th>
<th>ONSOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONCHAR</td>
<td>ONFILE</td>
<td>ONWCHAR</td>
</tr>
<tr>
<td>ONCODE</td>
<td>ONGSOURCE</td>
<td>ONWSOURCE</td>
</tr>
<tr>
<td>ONCONDCOND</td>
<td>ONKEY</td>
<td></td>
</tr>
<tr>
<td>ONCONDID</td>
<td>ONLOC</td>
<td></td>
</tr>
</tbody>
</table>
PL/I debugging techniques

For detailed information on these condition-handling built-in functions and pseudovariables, consult the PL/I Language Reference.

**ON-units for qualified and unqualified conditions**
There can only be one established ON-unit for an unqualified condition at any given point in a program, but there can be more than one established ON-unit for qualified conditions. For example, in handling the ENDFILE condition as qualified for different files, you can have an ON-unit established to uniquely handle the occurrence of ENDFILE for any one of the files.

**Conditions used for testing and debugging**
The following conditions are useful in testing and debugging your programs:
- SUBSCRIPTRANGE
- STRINGSIZE
- STRINGRANGE

Running your program with these conditions decreases performance, but ON-units for these conditions can serve as powerful tools for finding out the sources of errors in your program. You can enable any of these conditions by writing an ON-unit for them. Then, if the condition is raised, your ON-unit can define an action that tells you the cause of the error.

For example, if your program raises FIXEDOVERFLOW, it is useful to issue PUT DATA to discover the values of your data that led to the condition being raised.

In addition, the PREFIX option is useful because you can enable conditions without having to edit your program.

---

**Common programming errors**
A failure in running a PL/I program can be caused by:
- Logical errors in a source program
- Invalid use of PL/I (for example, uninitialized variables)
- Calling uninitialized entry variables
- Loops and other unforeseen errors
- Unexpected input/output data
- Unexpected program termination
- Other unexpected program results
- System failure
- Poor performance

**Logical errors in your source programs**
Logical errors in a source program are often difficult to detect and sometimes can make it appear as though there are compiler or library failures.

Some common errors in source programs are:
- Failure to convert correctly from arithmetic data
- Incorrect arithmetic and string-manipulation operations
- Failure to match data lists with their format lists

**Invalid use of PL/I**
A misunderstanding of the language can result in an apparent program failure. For example, any of the following programming errors can cause a program to fail:
- Using uninitialized variables
Common programming errors

- Using controlled variables that have not been allocated
- Reading records into incorrect structures
- Misusing array subscripts
- Misusing pointer variables
- Incorrect conversion
- Incorrect arithmetic operations
- Incorrect string-manipulation operations
- Freeing or using storage that was never allocated or already free

**Calling uninitialized entry variables**

If you call an entry variable that is uninitialized:

- Windows will raise a protection exception almost immediately.
- Windows 98, however, does not raise an immediate protection exception and allows you to execute instructions in low memory which can cause unpredictable program behavior.

**Loops and other unforeseen errors**

If an error is detected during execution of a PL/I program, and no ON-unit is provided in the program to terminate execution or attempt recovery, the job terminates abnormally. However, you can record the status of your program at the point where the error occurred by using an ERROR ON-unit that contains the statements:

```pli
on error
begin;
  on error system;
  call plidump ('TFBS','This is a dump');
end;
```

The statement ON ERROR SYSTEM; contained in the ON-unit ensures that further errors caused by attempting to transmit uninitialized variables do not result in an endless loop.

If you want to take action based on the specific type of condition being handled, use the ONCONDID function (for more information on this function, see the PL/I Language Reference):

```pli
on anycondition
begin;
  on anycondition system;
  select( oncondid() );
    when( condid_of1 )
      .
      .
      .
    when( condid_ufl )
      .
      .
    when( condid_zdiv )
      .
      .
    otherwise
      resignal;
  end;
end;
```
Common programming errors

Tips for dealing with loops
To prevent a permanent loop from occurring within an ON-unit, use the following code segment:

```plaintext
on Error begin;
  on Error System;
  .
  .
end;
```

If your program is caught in an endless loop, your primary concern is to be able to get out of the loop without shutting down your machine. The following solution is recommended for handling endless loops:

- When the loop is entered, press Ctrl-Break to end your program. No ATTENTION ON-unit is driven in this environment.

Unexpected input/output data
A program should contain checks to ensure that any incorrect input and output data is detected before it can cause the program to fail.

Use the COPY option of the GET and PUT statements if you want to check values obtained by stream-oriented input and output. The values are listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

Use the VALID built-in function to check the validity of PICTURE and FIXED DECIMAL identifiers.

For additional information on features that can lead to unexpected I/O, see Chapter 1, “Porting applications between platforms,” on page 3. Many of the features that can lead to portability problems (such as differences in ASCII and EBCDIC collating sequences) can also lead to unexpected I/O for your PL/I programs.

Unexpected program termination
If your program terminates abnormally without an accompanying run-time diagnostic message, the error that caused the failure probably also prevented the message from being displayed. Possible causes of this type of behavior are:

- Trying to run modules that were not compiled by this version of the compiler.
- Incorrect export DD: statements.
- Overwriting storage areas that contain executable instructions, particularly the PL/I communications area. Any of the following could cause overwriting of storage areas:
  - Assigning a value to a nonexistent array element. For example:
    ```plaintext
dcl array(10);
    .
    .
    do I = 1 to 100;
    array(I) = value;
    ```

    You can detect this type of error in a compile module by enabling the SUBSCRIPTRANGE condition. Each attempt to access an element outside the declared range of subscript values should raise the SUBSCRIPTRANGE condition. If there is no ON-unit for this condition, a diagnostic message prints and the ERROR condition is raised.
Common programming errors

Though this method is costly in terms of execution time and storage space, it is a valuable program testing aid. For more information on error handling, see “Using error and condition handling for debugging” on page 219.

– Using an incorrect locator value for a locator (pointer or offset) variable. This type of error is possible if a locator value is obtained using a record-oriented transmission.
Make sure that locator values created in one program, transmitted to a dataset, and subsequently retrieved for use in another program, are valid for use in the second program.

– Attempting to free a non-BASED variable. This can happen when you free a BASED variable after its qualifying pointer value has been changed. For example:
  ```pli
dcl a static,b based (p);
allocate b;
p = addr(a);
free b;
```

– Using an incorrect value for a label, entry, or file variable. Label, entry, and file values that are transmitted and subsequently retrieved are subject to the same kind of errors as those described previously for locator values.

– Using the SUBSTR pseudovariable to assign a string to a location beyond the end of the target string. For example:
  ```pli
dcl x char(3);
i = 3
substr(x,2,i) = 'ABC';
```

To detect this type of error in a compiled module, use the STRINGRANGE condition (for more information, see “Conditions used for testing and debugging” on page 222).

Other unexpected program results

Because Windows responds to floating-point conditions differently, you might experience altered program flow. One consequence of altered program flow is conditions that do not get raised because they have become disabled.

For example, although using the NOIMPRECISE compile-time option does provide better floating-point error detection than IMPRECISE, the Windows operating system does not always detect floating-point exceptions immediately. If you have a statement in your program that is likely to raise a floating-point exception, you can avoid this detection problem by enclosing the statement, by itself, in a BEGIN block.

Compiler or library subroutine failure

If you are convinced that the failure is caused by a compiler failure or a library subroutine failure, you should contact IBM.

Meanwhile, you can attempt to find an alternative way to perform the operation that is causing the trouble. A bypass is often possible because the PL/I language frequently provides an alternative method of performing a given operation.

System failure

System failures include machine malfunctions and operating system errors. System messages identify these failures to the operator.
Common programming errors

**Poor performance**

While not necessarily caused by bugs, poor performance is associated with excessive run-time and memory requirements. One thing to keep in mind is that many debugging techniques (such as enabling SUBSCRIPTRANGE) tend to decrease performance.

One feature that can increase performance is the OPTIMIZE compile-time option (see “OPTIMIZE” on page 99). For additional information on improving program performance, see Chapter 17, “Improving performance,” on page 355.
### Part 4. Input and output

#### Chapter 10. Using data sets and files

- Types of data sets
  - Native data sets
  - Conventional text files and devices
  - Fixed-length data sets
  - Additional data sets
  - Varying-length data sets
  - Regional data sets
  - Workstation VSAM data sets

- Establishing data set characteristics
  - Records
  - Record formats
  - Data set organizations
  - Specifying characteristics using the PL/I `ENVIRONMENT` attribute
    - `CONSECUTIVE`
    - `CTRLASA`
    - `GENKEY`
    - `GRAPHIC`
    - `KEYLENGTH`
    - `KEYLOC`
    - `ORGANIZATION`
    - `RECSIZE`
    - `REGIONAL(1)`
    - `SCALARVARYING`
    - `VSAM`

- Specifying characteristics using DD:ddname environment variables
  - `AMTHD`
  - `APPEND`
  - `ASA`
  - `AUTOPAGE`
  - `BUFSIZE`
  - `CHARSET for record I/O`
  - `CHARSET for stream I/O`
  - `DELAY`
  - `DELIMIT`
  - `LRECL`
  - `LRSKIP`
  - `PROMPT`
  - `PUTPAGE`
  - `RECCOUNT`
  - `RECSIZE`
  - `RETRY`
  - `SAMELINE`
  - `SHARE`
  - `SKIP0`
  - `TERMLBUF`
  - `TYPE`

- Opening a file
- Closing a file
- Associating several data sets with one file
- Combinations of I/O statements, attributes, and options
- `DISPLAY` statement input and output
- `PL/I` standard files (SYSPRINT and SYSIN)
- Redirecting standard input, output, and error devices

- Chapter 11. Defining and using consecutive data sets
  - Printer-destined files
  - Using stream-oriented data transmission
    - Defining files using stream I/O
    - `ENVIRONMENT` options for stream-oriented data transmission
    - Creating a data set with stream I/O
      - Essential information
    - Example
    - Accessing a data set with stream I/O
      - Essential information
      - Example
    - Using PRINT files
      - Controlling printed line length
      - Overriding the tab control table
    - Using SYSIN and SYSPRINT files
  - Controlling input from the terminal
    - Using files conversationally
    - Format of data
    - Stream and record files
    - Capital and lowercase letters
    - End of file
  - Controlling output to the terminal
    - Format of PRINT files
    - Stream and record files
    - Example of an interactive program
  - Using record-oriented I/O
    - Defining files using record I/O
    - `ENVIRONMENT` options for record-oriented data transmission
    - Creating a data set with record I/O
      - Essential information
    - Accessing and updating a data set with record I/O
      - Essential information
    - Examples of consecutive data sets

- Chapter 12. Defining and using regional data sets
  - Defining files for a regional data set
  - Specifying `ENVIRONMENT` options
  - Essential information for creating and accessing regional data sets
  - Using keys with regional data sets
  - Using `REGIONAL(1)` data sets

© Copyright IBM Corp. 1998, 2012
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy records</td>
<td>280</td>
</tr>
<tr>
<td>Creating a REGIONAL(1) data set</td>
<td>280</td>
</tr>
<tr>
<td>Example</td>
<td>281</td>
</tr>
<tr>
<td>Accessing and updating a REGIONAL(1) data set</td>
<td>282</td>
</tr>
<tr>
<td>Sequential access</td>
<td>282</td>
</tr>
<tr>
<td>Direct access</td>
<td>283</td>
</tr>
<tr>
<td>Example</td>
<td>283</td>
</tr>
</tbody>
</table>

**Chapter 13. Defining and using workstation VSAM data sets**

- Moving data between the workstation and mainframe                   | 287  |
- Workstation VSAM organization                                       | 287  |
  - Creating and accessing workstation VSAM data sets                   | 288  |
  - Determining which type of workstation VSAM data set you need        | 288  |
  - Accessing records in workstation VSAM data sets                     | 288  |
  - Using keys for workstation VSAM data sets                           | 289  |
  - Using keys for workstation VSAM keyed data sets                     | 289  |
  - Using sequential record values                                      | 290  |
  - Using relative record numbers                                       | 290  |
- Choosing a data set type                                              | 290  |
- Defining files for workstation VSAM data sets                        | 292  |
  - Specifying options of the PL/I ENVIRONMENT attribute                | 292  |
  - Adapting existing programs for workstation VSAM                     | 293  |
    - Adapting programs using CONSECUTIVE files                          | 293  |
    - Adapting programs using INDEXED files                              | 293  |
    - Adapting programs using REGIONAL(1) files                          | 294  |
    - Adapting programs using VSAM files                                 | 294  |
- Using workstation VSAM sequential data sets                          | 295  |
  - Using a sequential file to access a workstation VSAM sequential data set | 296  |
  - Defining and loading a workstation VSAM sequential data set          | 296  |
  - Updating a sequential data set                                      | 298  |
- Workstation VSAM keyed data sets                                     | 298  |
  - Loading a workstation VSAM keyed data set                           | 300  |
  - Using a SEQUENTIAL file to access a workstation VSAM keyed data set | 302  |
  - Using a DIRECT file to access a workstation VSAM keyed data set     | 302  |
- Workstation VSAM direct data sets                                    | 305  |
  - Loading a workstation VSAM direct data set                          | 307  |
  - Using a SEQUENTIAL file to access a workstation VSAM direct data set | 309  |
  - Using READ statements                                                | 309  |
  - Using WRITE statements                                               | 309  |
  - Using the REWRITE or DELETE statements                               | 309  |
  - Using a DIRECT file to access a workstation VSAM direct data set    | 309  |
Chapter 10. Using data sets and files

Your PL/I programs can process and transmit units of information called records. A collection of records is called a data set, but for PL/I workstation products, a data set can be either a file or a device. Data sets are logical collections of information external to PL/I programs; they can be created, accessed, or modified by programs written in PL/I.

Your PL/I program recognizes and processes information in a data set by associating it with a symbolic representation of the data set called a PL/I file. This PL/I file represents the environment independent characteristics of a set of input and output operations.

In order to minimize confusion, this book uses the term PL/I file to refer to the file declared and used in a PL/I program. The terms data set and workstation file (or workstation device) are used to refer to the collection of data on an external I/O device. In some cases the data sets have no name; they are known to the system by the device on which they exist.

Types of data sets

PL/I defines the following two types of data sets:

- Native data set. It is a PL/I term used to define conventional text files and devices associated with the platform in use.
- Workstation VSAM data set. It is used to refer to files that are similar to mainframe VSAM data sets. PL/I uses either the DDM, ISAM, or BTRIEVE access method to create and access these types of data sets.

Platform distinctions

This chapter refers to the access methods available on PL/I workstation products; however, all methods are not available on all platforms. As you refer to information in this chapter, use the following guideline:

- DDM—supported on AIX only
- ISAM—supported on AIX and Windows
- BTRIEVE—supported on Windows only
- REMOTE—supported on Windows to access mainframe data files

To convert mainframe VSAM files to the corresponding DDM, ISAM, or BTRIEVE files, follow the procedure documented in the prolog for the LODVSAM utility (not yet supported on AIX). Make sure you specify the appropriate access method AMTHD(DDM|ISAM|BTRIEVE).

To convert DDM, ISAM, or BTRIEVE files to corresponding mainframe VSAM files, follow the procedure documented in the prolog for the RELOAD utility (not yet supported on AIX). These utilities should be in the samples directory for PL/I for Windows.

Data sets that reside on the mainframe can be accessed remotely by your PL/I program using the Distributed FileManager product that comes with SMARTdata Utilities (SdU), one of the PL/I for Windows components. You can find
Types of data sets

information about using SdU in the online books for that product. The online
books for SdU are installed only if you select that component.

For Windows, refer to the Distributed FileManager User’s Guide.

There are several types of native data sets:
- Conventional text files
- Character devices
- Fixed-length data sets

Both record and stream I/O can be used to access these types of data sets, which
can be accessed only in a sequential manner.

Additional types of PL/I-defined data sets include:
- Varying-length
- Regional
- Workstation VSAM data sets

Only record I/O can be used to access regional data sets. Access can be either
sequential or direct.

Native data sets

A native data set in PL/I terms defines conventional text files and devices
associated with the platform you are using.

Conventional text files and devices

A conventional text file has logical records delimited by the CR - LF (carriage
return and line feed) character sequence. Most text editor programs create, and
allow you to alter, conventional text files. Your PL/I programs can create
conventional text files, or they can access text files that were created by other
programs.

Devices for workstation products are the keyboard, screen, and printer. The names
you use to refer to them in PL/I are:

In a non-interactive environment, SYSIN and SYSPRINT can be pipes or files.

**NUL**: (or NUL)
Null output device (for discarding output)

**STDIN**:  
Standard input file (defaults to CON)

**STDOUT**:  
Standard output file (defaults to CON)

**STDERR**:  
Standard error message file (defaults to CON)

**Note**: STDIN:, STDOUT:, and STDERR: can be redirected, whereas the other device
names cannot.

Fixed-length data sets

PL/I also allows you to treat a file as a set of fixed-length records. Your PL/I
programs can create fixed-length data sets, or access existing files as fixed-length
data sets. The data access does not treat Carriage Return(CR) or Line Feed (LF) as
characters with special meaning. In particular, the CR - LF sequence does not
delimit records, although these characters can be contained in the data set. It is the
length you specify that determines what PL/I considers to be a record within the
Types of data sets

data set. This type of data set has the restriction that the total number of characters in the data set must be evenly divisible by the length you specify.

Fixed-length data sets can be accessed only in a sequential manner.

Additional data sets

Other types of data sets include varying-length, regional, and workstation VSAM data sets.

Varying-length data sets

Your PL/I program can also create and access data sets where each record has a 2-byte prefix that specifies the number of bytes in the rest of the record. Unlike files with records delimited by CR - LF, these varying-length files can have records that possibly contain arbitrary bit patterns.

Regional data sets

A description of regional data sets and how you can use them is presented in Chapter 12, “Defining and using regional data sets,” on page 277.

Note: Regional in this context means the same thing as REGIONAL(1) does in PL/I for z/OS.

Workstation VSAM data sets

The PL/I workstation products support VSAM file organization. There are three types of VSAM data sets on the workstation:
- Consecutive, similar to a VSAM entry-sequenced data set (ESDS)
- Relative, similar to a VSAM relative record data set (RRDS)
- Indexed, similar to a VSAM key-sequenced data set (KSDS)

The PL/I workstation products currently support the following methods for accessing VSAM data sets:
- ISAM (AIX and Windows)
- BTRIEVE (Windows only)
- REMOTE to access mainframe data files on Windows

DDM access method

DDM data sets are record-oriented files as defined by the SMARTdata Utilities (SdU) architecture. Workstation VSAM data sets that use the DDM access method can exist on local systems. You can compile and run most existing mainframe programs that reference mainframe VSAM data sets. A DDM keyed data set is represented by two files—one called the base, and the other called the prime index. The records of the data set are kept in the base; the prime index contains information about the primary keys of the data set. When you create a DDM keyed data set, you specify the name of the base; DDM generates a name for the prime index, which it derives from the name of the base.

When you use DDM data sets, you do not need to be concerned about record length, except that your records cannot exceed the maximum specified length.

You can compile and run most existing mainframe programs that reference mainframe VSAM data sets by creating the appropriate workstation VSAM data set on your PC before running the program.

ISAM access method

Unless otherwise specified, the term ISAM in this chapter refers to the STL
Types of data sets

(Standard Language) file system, which is a local access method and not mainframe ISAM. ISAM data sets are stored in one file and can exist on local file systems only.

**SFS access method**

The term SFS in this chapter refers to Encina SFS, which is one of the file systems used by CICS.

**BTRIEVE access method (Windows only)**

The BTRIEVE access method is provided to allow you to use PL/I input and output statements to access files created under CICS. There is currently no PL/I support for BTRIEVE segmented and multiple keys.

BTRIEVE data sets are stored in one file and can exist on local file systems only.

**REMOTE access method on Windows**

The REMOTE access method is provided to allow you to remotely access data files on the mainframe.

Detailed information on workstation VSAM is found in Chapter 13, “Defining and using workstation VSAM data sets,” on page 287.

Establishing data set characteristics

When you declare or open a file in your program, you are describing to PL/I the characteristics of the file. You can also use a DD:ddname environment variable or an expression in the TITLE option of the OPEN statement to describe to PL/I the characteristics of the data in data sets or in PL/I files associated with them. See “Associating a PL/I file with a data set” on page 248 for more information.

You do not always need to describe your data both within the program and outside it; often one description serves for both data sets and their associated PL/I files. There are, in fact, advantages to describing your data's characteristics in only one place. These are described later in this chapter and in following chapters.

To effectively describe your program data and the data sets you are using, you need to understand something about how PL/I moves and stores data.

Records

A record is the unit of data transmitted to and from a program. You can specify the length of records in the RECSIZE option for any of the following:

- **DD:ddname** environment variable
- PL/I ENVIRONMENT attribute
- TITLE option of the OPEN statement

Except for certain stream files, where defaults are applied, you must specify the RECSIZE option when your PL/I program creates a data set. For more information about stream files, see Chapter 11, “Defining and using consecutive data sets,” on page 255.

You must also specify the RECSIZE option when your program accesses a data set that was not created by PL/I.

Please note that an editor might alter a data set implicitly. You should use special caution if you examine a non CR - LF file using an editor, because most editors automatically insert CR - LF or similar character sequences.
Establishing data set characteristics

Record formats
The records in a data set can have one of the following formats:
  - Undefined-length
  - Fixed-length
  - Varying-length

For a native file, you specify either undefined-length or fixed-length record format in the TYPE option of the DD information. You do not need to specify a record format for workstation VSAM data sets; they implicitly consist of varying-length records.

Data set organizations
The options of the PL/I ENVIRONMENT attribute that specify data set organization are:
  - CONSECUTIVE
  - ORGANIZATION(CONSECUTIVE)
  - ORGANIZATION(INDEXED)
  - ORGANIZATION(RELATIVE)
  - REGIONAL(1)
  - VSAM

Each is described in "Specifying characteristics using the PL/I ENVIRONMENT attribute."

If you do not specify the data set organization option in the ENVIRONMENT attribute, it defaults to CONSECUTIVE.

Specifying characteristics using the PL/I ENVIRONMENT attribute
The ENVIRONMENT attribute of the DECLARE statement allows you to specify certain data set characteristics within your programs. These characteristics are not part of the PL/I language; hence, using them in a file declaration might make your program non-portable to other PL/I implementations.

Here is an example of how to specify environment options for a file in your program:
```
  declare Invoices file environment(regional(1), recsize(64));
```

The options you can specify in the ENVIRONMENT attribute are defined in the following sections.

CONSECUTIVE
The CONSECUTIVE option defines a file with consecutive data set organization. In a data set with CONSECUTIVE organization, 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.

You use the CONSECUTIVE option to access native data sets using either stream-oriented or record-oriented data transmission. You also use it for input files
Establishing data set characteristics

declared with the SEQUENTIAL attribute and associated with a workstation
VSAM data set. In this case, records in a workstation VSAM keyed data set are
presented in key sequence.

CONSECUTIVE is the default data set organization.

**CTLASA**
The CTLASA option specifies that the first character of a record is to be interpreted
as an American National Standard Institute (ANSI) print control character. The
option applies only to RECORD OUTPUT files associated with consecutive data
sets.

```
CONTROLA
```

The ANSI print control characters, listed in Table 11 on page 255, cause the
specified action to occur before the associated record is printed.

For information about how you use the CTLASA option, see “Printer-destined
files” on page 255.

The IBM Proprinter control characters require up to 3 bytes more than the single
byte required by an ANSI printer control character. However, do not adjust your
logical record length specification (see the RECSIZE environment option) because
PL/I automatically adds 3 to the logical record length when you specify CTLASA.

You can modify the effect of CTLASA so that the first character of records is left
untranslated to IBM Proprinter control characters. See the ASA environment option
under “ASA” on page 240.

Do not specify the SCALARVARYING environment option for printer-destined
output operations, as PL/I does not know how to interpret the first data byte of
records.

**GENKEY**
The GENKEY (generic key) option applies only to workstation VSAM indexed data
sets. It enables you to classify keys recorded in the data set and to use a
SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access
records according to their key class.

```
GENKEY
```

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.
Establishing data set characteristics

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 having 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, since 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:

```plaintext
dcl ind file record sequential keyed
   update env (indexed 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 with a key beginning '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 could result in reading records from higher key classes, since 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 would 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 KEYLENGTH subparameter. The KEYLENGTH 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.
Establishing data set characteristics

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 the only member of its class).

**GRAPHIC**
You must specify the GRAPHIC option if you use DBCS variables or DBCS constants in GET and PUT statements for list-directed and data-directed I/O. You can also specify the GRAPHIC option for edit-directed I/O.

```plaintext
   GRAPHIC
```

PL/I raises the ERROR condition for list-directed and data-directed I/O if you have graphics in input or output data and you do not specify the GRAPHIC option.

For information on the graphic data type, and on the G-format item for edit-directed I/O, see the *PL/I Language Reference*.

**KEYLENGTH**
The KEYLENGTH option specifies the length, \( n \), of the recorded key for a KEYED file. You can specify KEYLENGTH only for INDEXED files (see ORGANIZATION later in this section).

```plaintext
   KEYLENGTH\( (n) \)
```

If you include the KEYLENGTH option in a file declaration, and the associated data set already exists, the value is used for checking purposes. If the key length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

**ISAM and BTRIEVE**
Keys are kept in the index pages of an ISAM or BTRIEVE file. The length of the key needs to be defined to PL/I when the file is created.

**KEYLOC**
The KEYLOC option specifies the starting position, \( n \), of the embedded key in records of a KEYED file. You can specify KEYLOC only for INDEXED files (see ORGANIZATION later in this section).

```plaintext
   KEYLOC\( (n) \)
```

The position, \( n \), must be within the limits:
\[ 1 \leq n \leq \text{recordsize} - \text{keylength} + 1 \]
Establishing data set characteristics

That is, the key cannot be larger than the record and must be contained completely within the record.

This means that if you specify the SCALARVARYING option, the embedded key must not overlap the first two bytes of the record; hence, the value you specify for KEYLOC must be greater than 2.

If you do not specify KEYLOC when creating an indexed data set, the key is assumed to start with the first byte of the record.

If you include the KEYLOC option in a file declaration, and the associated data set already exists, the value is used for checking purposes. If the key position you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

**ISAM and BTRIEVE**

Keys are kept in the index pages of an ISAM or BTRIEVE file. The location of the key needs to be defined to PL/I when the file is created.

**ORGANIZATION**

The ORGANIZATION option specifies the organization of the data set associated with the PL/I file.

```plaintext
ORGANIZATION
  CONSECUTIVE
  INDEXED
  RELATIVE
```

**CONSECUTIVE**

Specifies that the file is associated with a consecutive data set. A consecutive file may be either a native data set or a workstation VSAM sequential, direct, or keyed data set.

**INDEXED**

Specifies that the file is associated with an indexed data set. INDEXED specifies that the data set contains records arranged in a logical sequence, according to keys embedded in each record. Logical records are arranged in the data set in ascending key sequence according to the ASCII collating sequence. An indexed file is a workstation VSAM keyed 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 workstation VSAM direct data set. Relative keys range from 1 to mnnn.

**RECSIZE**

The RECSIZE option specifies the length, \( n \), of records in a data set.

```plaintext
RECSIZE \( n \)
```
Establishing data set characteristics

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 records can have.

If you include the RECSIZE option in a file declaration, and the file is associated with a workstation VSAM data set that already exists, the value is used for checking purposes. If the record length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

Specify the RECSIZE option when you access data sets created by non-PL/I programs such as text editors.

**ISAM and BTRIEVE**
You must specify RECSIZE when using the BTRIEVE or ISAM access method.

**REGIONAL(1)**
The REGIONAL(1) option defines a file with the regional organization.

A data set with regional organization contains fixed-length 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).

For information about how you use regional data sets, see Chapter 12, “Defining and using regional data sets,” on page 277.

**SCALARVARYING**
The SCALARVARYING option is used in the input and output of VARYING strings.

When storage is allocated for a VARYING 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 you specify SCALARVARYING for the file.

When you use locate mode statements (LOCATE and READ SET) to create and read a data set with element VARYING strings, you must specify SCALARVARYING to indicate that a length prefix is present, since the pointer that locates the buffer is always assumed to point to the start of the length prefix.

When you specify this option and element VARYING strings are transmitted, you must allow 2 bytes in the record length to include the length prefix.

A data set created using SCALARVARYING should be accessed only by a file that also specifies SCALARVARYING.
Establishing data set characteristics

SCALARVARYING and CTLASA must not be specified for the same file, as this causes the first data byte to be ambiguous.

VSAM
The VSAM option is provided for compatibility with PL/I for z/OS.

Specifying characteristics using DD:ddname environment variables

You use the SET 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.

The syntax of the DD:ddname environment variable is:

```
DD:ddname=filespec,option
```

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**

Specifies the name of the environment variable. The ddname can be either the name of a file constant or an alternate ddname that you specify in the TITLE option of your OPEN statement. The TITLE option is described in “Using the TITLE option of the OPEN statement” on page 249.

If you use an alternate ddname, and it is longer than 31 characters, only the first 31 characters are used in forming the environment variable name.

**option**

The options that you can specify as DD information are described in the pages that follow, beginning with "AMTHD" and ending with "TYPE" on page 246.

**AMTHD**
The AMTHD option specifies the access method to access the data set.
Establishing data set characteristics

**FSYS**
Specifies that PL/I uses its native access methods to access a native file. This is the default.

**ISAM**
Specifies that the ISAM access method is to be used to access an ISAM file.

**BTRIEVE (Windows)**
Specifies that the BTRIEVE access method is to be used to access a BTRIEVE file.

**REMOTE (Windows)**
Specifies that the file resides on a remote DDM target system (such as MVS).

For Windows, the name of the file needs to be qualified by the LU alias or the fully-qualified SNA network name.

FSYS is used by default if you do not specify the AMTHD option and if you do not apply one of the following ENVIRONMENT options:
- ORGANIZATION(INDEXED)
- ORGANIZATION(RELATIVE)
- VSAM

If you specify any of the above options, AMTHD(ISAM) is the default on Windows while AMTHD(DDM) is the default on AIX.

**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. This is the default.

N Specifies that, if the file exists, it is to be re-created.

The APPEND option applies only to OUTPUT files. APPEND is ignored if:
- The file does not exist
- The file does not have the OUTPUT attribute
- The organization is REGIONAL(1)

**ASA**
The ASA option applies to printer-destined files. This option specifies when the ANSI control character in each record is to be interpreted.

N Specifies that the ANSI print control characters are to be translated to IBM Proprinter control characters when records are written to the data set. This is the default.
Establishing data set characteristics

Y  Specifies that the ANSI print control characters are not to be translated; instead they are to be left as is for subsequent translation by a process you determine.

If the file is not a printer-destined file, the option is ignored. Printer-destined files are described in “Printer-destined files” on page 255.

AUTOPAGE
The AUTOPAGE option specifies whether a form feed character is the first control character output to a PRINT file, regardless of the PUT option used.

Y  Specifies that the library puts a form feed control character as the first control character in a PRINT file, even if the first I/O statement to the PRINT file is a PUT SKIP statement. AUTOPAGE(Y) is the default.

N  Specifies that the library interprets the PUT option used. For example, if the first I/O statement to the PRINT file is a PUT SKIP statement, the library interprets the SKIP option; that is, a carriage return control character is used.

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, and has a default value for BUFSIZE of zero.

If the value of zero is given to BUFSIZE, 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 gives the user the capability of using EBCDIC data files as input files, and specifying the character set of output files.
Establishing data set characteristics

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file have (output).

CHARSET(ASIS) is the default.

**CHARSET for stream I/O**

This version of the CHARSET option applies for stream input and output files. It gives the user the capability of using EBCDIC data files as input files, and specifying 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 ASCII.

\[ \text{CHARSET(ASIS)} \]

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file to have (output).

CHARSET(ASCII) is the default.

**DELAY**

The DELAY option specifies the number of milliseconds to delay before retrying an operation that fails when a file or record lock cannot be obtained by the system.

\[ \text{DELAY}(n) \]

The default value for DELAY is 0.

**DELIMIT**

The DELIMIT option specifies whether the input file contains field delimiters or not. 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.

\[ \text{DELIMIT}(\text{Y}) \]

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.
Establishing data set characteristics

**LRECL**
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 commence on the nth (n refers to the value specified with the SKIP option of the PUT or GET statement) line of the first page for the first SKIP format item to be executed after a file is opened.

If n is zero or 1, output commences on the first line of the first page.

**PROMPT**
The PROMPT option specifies whether or not colons should be visible as prompts for stream input from the terminal.

PROMPT(N) is the default.

**PUTPAGE**
The PUTPAGE option specifies whether or not the form feed character should be followed by a carriage return character. This option only applies 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). This is the default.

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.
Establishing data set characteristics

**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.

```
RECCOUNT(n)
```

The RECCOUNT option is ignored if PL/I does not create, or re-create, the data set. If the RECCOUNT option applies and is omitted, the default is 50 for regional and relative files.

**RECSIZE**
The RECSIZE option specifies the length, \( n \), of records in the data set.

```
RECSIZE(n)
```

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 records may have.

The default for \( n \) is 512.

**RETRY**
The RETRY option specifies the number of times an operation should be retried when a file or record lock cannot be obtained by the system.

```
RETRY(n)
```

The default value for RETRY is 10. This option is applicable only to DDM files.

**SAMELINE**
The SAMELINE option specifies whether the system prompt occurs on the same line as the statement that prompts for input.

```
SAMELINE(N)
```

The following examples show the results of certain combinations of the PROMPT and SAMELINE options:
Establishing data set characteristics

Example 1

Given the statement `PUT SKIP LIST('ENTER:');`, output is as follows:

<table>
<thead>
<tr>
<th>prompt(y), sameline(y)</th>
<th>ENTER: (cursor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prompt(n), sameline(y)</td>
<td>ENTER: (cursor)</td>
</tr>
<tr>
<td>prompt(y), sameline(n)</td>
<td>ENTER: (cursor)</td>
</tr>
<tr>
<td>prompt(n), sameline(n)</td>
<td>ENTER: (cursor)</td>
</tr>
</tbody>
</table>

Example 2

Given the statement `PUT SKIP LIST('ENTER');`, output is as follows:

<table>
<thead>
<tr>
<th>prompt(y), sameline(y)</th>
<th>ENTER: (cursor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prompt(n), sameline(y)</td>
<td>ENTER: (cursor)</td>
</tr>
<tr>
<td>prompt(y), sameline(n)</td>
<td>ENTER: (cursor)</td>
</tr>
<tr>
<td>prompt(n), sameline(n)</td>
<td>ENTER: (cursor)</td>
</tr>
</tbody>
</table>

SHARE

The SHARE option specifies the level of file sharing to be allowed.

```
SHARE(READ)
```

NONE

Specifies that the file is not to be shared with other processes. This is the default.

READ

Specifies that other processes can read the file.

ALL

Specifies that other processes can read or write the file. Data integrity is the user's responsibility, and PL/I provides no assistance in maintaining it.

This option is valid only with DDM files.

To enable record-level locking, specify SHARE(ALL) and declare the file as an update file. This is recommended when running CICS applications.

The UNDEFINEDFILE condition is raised if the requested or default level of file sharing cannot be obtained.

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.
Establishing data set characteristics

**SKIP0(N)**
Specifies that the cursor is to be moved to the beginning of the next line. This is the default.

**SKIP0(Y)**
Specifies that the cursor to be moved 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:
```
set dd:sysprint=stdout:,SKIP0(Y)
set dd:sysprint=con,SKIP0(Y)
```

**TERMLBUF**
The TERMLBUF option specifies the maximum number of lines in the window of a PL/I Presentation Manager (PM) terminal.

```
TERMLBUF-(n)
```

If the file is not associated with a PM terminal, the option is ignored. The default is 512 lines.

**TYPE**
The TYPE option specifies the format of records in a native file.

```
TYPE-(CRLF
LF
TEXT
FIXED
VARLS
VARLS4X4
VARMS4X4
VARMS
LL
LLLS
LLZZ
CRLFEOL)
```

**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. See restrictions on file attributes). 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.
Establishing data set characteristics

The data set must not contain any record that is longer than the value determined for the record length of the data set.

This is the default for ISAM and BTRIEVE.

**LF**

Specifies that records are delimited by the LF character combination. ('LF' represents the ASCII values of feed or '0A'x. See restrictions on file attributes)

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 CRLF.

**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.

**VARLS|VARMS**

Indicates that records have a 2-byte prefix that specifies the number of bytes in the rest of the record.

For VARLS, the length prefix is in little endian format. These records look like NATIVE CHAR VARYING strings.

For VARMS, the length prefix is in big endian format. These records look like NONNATIVE CHAR VARYING strings.

TYPE(VARLS) data sets provide the fastest way to use PL/I to read and write data sets containing records of variable length and arbitrary byte patterns. This is not possible with TYPE(CRLF) data sets because when a record is read that was written containing the bit string '0d0a'b4, a misinterpretation occurs.

TYPE(VARMS) data sets provide a way to read SCALARVARYING files downloaded from the mainframe.

**VARLS4X4|VARMS4X4**

Indicates that records have a 4-byte prefix and a 4-byte suffix. The prefix and suffix each contain the number of bytes in the rest of the record. This number does not include either the 4 bytes used by the prefix or the 4 bytes used by the suffix.

For VARLS4X4, the data is in little endian format, which is used on a PC. For VARMS4X4, the data is in big endian format, which is downloaded from the z/OS or AIX system.

Type(VARLS4X4) and Type(VARMS4X4) data sets provide a way to handle FORTRAN sequential unformatted files.

**LL|LLLL**

Indicates that records have a 2-byte prefix that specifies the total number of bytes in the record (including the prefix).
Establishing data set characteristics

TYPE(LL) data sets provide a way to read files downloaded from the mainframe with a tool (see VRECGEN.PL1 sample program) that appends 2 bytes. The length is in big endian format.

TYPE(LLL) data sets provide a way to read files downloaded from the PC. The length is in little endian format.

LLZZ
Specifies that records have a 4-byte prefix held the same way as varying records on S/390.

The LLZZ suboption provides a way to read and write data sets that contain records of variable length and arbitrary byte patterns. This does not apply to TYPE(CRLF) data sets. Under CRLF, a written record containing the bit string '0d0a'b4 is misinterpreted when it is read.

A TYPE(LLZZ) data set must not contain any record that is longer than the value determined for the record length of the data set.

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 using non-print files with the TYPE(FIXED) attribute, SKIP is replaced by trailing blanks to the end of the line. If TYPE(CRLF) is being used, SKIP is replaced by CRLF with no trailing blanks.

Associating a PL/I file with a data set

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 set(s) 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.

Using environment variables
You use the SET 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.
Associating a PL/I file with a data set

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), for example:

```
declare MyFile stream output;
```

You can specify options for the SET command by including them on the command line.

```
set dd:myfile=c:\ datapath\mydata.dat,APPEND(N)
```

If you are familiar with the IBM mainframe environment, you can think of the environment variable much like you do the:
- DD statement in MVS
- ALLOCATE statement in TSO
- FILEDEF command in CMS

For more about the syntax and options you can use with the DD:ddname environment variable, see “Specifying characteristics using DD:ddname environment variables” on page 239.

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.

```
TITLe(expression)
```

The expression must yield a character string with the following syntax:

```
alternate_ddname/filespec,-dd_option-
```

**alternate_ddname**

The name of an alternate DD:ddname environment variable. An alternate DD:ddname environment variable is one not named after a file constant. For example, if you had a file named INVENTRY in your program, and you establish two DD:ddname environment variables—the first named INVENTRY and the second named PARTS—you could associate the file with the second one using this statement:

```
open file(INVENTRY) title('PARTS');
```

**filespec**

Any valid file specification on the system you are using.

**dd_option**

“Specifying characteristics using DD:ddname environment variables” on page 239. One or more options allowed in a DD:ddname environment variable. For more about options of the DD:ddname environment variable, see “Specifying characteristics using DD:ddname environment variables” on page 239.
Associating a PL/I file with a data set

Here is an example of using the OPEN statement in this manner:

```
open file(Payroll) title('/June.Dat,append(n),recsize(52)');
```

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.

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 the CON device.

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 in the DPATH environment variable.

Opening and closing PL/I files

This topic summarizes what PL/I does when your application executes the OPEN and CLOSE statements.

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. The information is merged using the following order of precedence:
1. Attributes on the OPEN statement
2. ENVIRONMENT options on a file declaration
3. Values in TITLE option of the OPEN statement when '/ ' is used
4. Values in the DD:ddname environment variable
5. IBM defaults

When the data set being opened is not a workstation device, the paths specified in the DPATH environment variable are searched for the data set. If the data set is not found, and the file has the OUTPUT attribute, the data set is created in the current directory.

If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Closing a file

The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated.

Associating several data sets with one file

A PL/I file can, at different times, represent entirely different data sets. The TITLE option allows you to choose dynamically, at open time, among several data sets to be associated with a particular PL/I file. Consider the following example:
In this example, when Master is opened during the first iteration of the do-group, the file is associated with the data set named MASTER1A.DAT. After processing, the file is closed, dissociating the PL/I file MASTER from the MASTER1A.DAT data set. During the second iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the data set named MASTER1B.DAT. Similarly, during the final iteration of the do-group, MASTER is associated with the data set MASTER1C.DAT.

**Combinations of I/O statements, attributes, and options**

The figures that follow list the I/O statements, file attributes, ENVIRONMENT options, and DD:ddname environment variable options you can use for the various PL/I file operations. Table 9 lists those for native data sets and Table 10 on page 252 lists those for workstation VSAM data sets.

**Table 9. Statements, attributes, and options for native data sets**

<table>
<thead>
<tr>
<th>Statements</th>
<th>File attributes</th>
<th>ENVIRONMENT options</th>
<th>DD_DDNAME options</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>ENVIRONMENT</td>
<td>CONSECUTIVE</td>
<td>AMTHD(FSYS)</td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td>GRAPHIC</td>
<td>APPEND(Y</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>RECSIZE(n)^1</td>
<td>ASA(Y</td>
</tr>
<tr>
<td></td>
<td>PRINT</td>
<td></td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>STREAM</td>
<td></td>
<td>RECSIZE(n)^1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TERMLBUF(n)^3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TYPE(CRLF</td>
</tr>
</tbody>
</table>

| GET        | ENVIRONMENT     | CONSECUTIVE         | AMTHD(FSYS)       |
|            | FILE            | GRAPHIC             | file_spec         |
|            | STREAM          | RECSIZE(n)^1        | RECSIZE(n)^4      |
|            | INPUT           |                     | SHARE(NONE|READ|ALL) |
|            |                 |                     | TERMLBUF(n)^3     |
|            |                 |                     | TYPE(CRLF|TEXT|FIXED) |

| WRITE      | BUFFERED | CONSECUTIVE | AMTHD(FSYS) |
| DIRECT     | UNBUFFERED | REGIONAL(1)  | APPEND(Y|N)   |
| SEQUENTIAL |          | CTLSA^2      | file_spec    |
| ENVIRONMENT|          | RECSIZE(n)^1 | RECSIZE(n)^1 |
| FILE       |          | SCALARVARYING | SHARE(NONE|READ|ALL) |
| KEYED^a    |          |              | TERMLBUF(n)^3 |
| RECORD     |          |              | TYPE(CRLF|TEXT|FIXED)^2 |
| OUTPUT     |          |              |               |
| UPDATE     |          |              |               |

| LOCATE     | BUFFERED | CONSECUTIVE | AMTHD(FSYS) |
| ENVIRONMENT|          | REGIONAL(1) | APPEND(Y|N)  |
| FILE       |          | CTLSA^2     | file_spec   |
| KEYED^a    |          | RECSIZE(n)^1| RECSIZE(n)^1 |
| RECORD     |          | SHARE(NONE|READ|ALL) |
| OUTPUT     |          |              | TYPE(CRLF|TEXT|FIXED)^2 |
| SEQUENTIAL |          |              |               |

Chapter 10. Using data sets and files 251
## Statements, attributes, options

**Table 9. Statements, attributes, and options for native data sets (continued)**

<table>
<thead>
<tr>
<th>Statements</th>
<th>File attributes</th>
<th>ENVIRONMENT options</th>
<th>DD_DDNAME options</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>BUFFERED</td>
<td>CONSECUTIVE</td>
<td>AMTHD(FSYS)</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td>REGIONAL(1)</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>RECSIZE(n)</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>ENVIROMENT</td>
<td>VARYING</td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>INPUT UPDATE</td>
<td></td>
<td>TERMLBUF(n)</td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td>SCALARVARYING</td>
<td>TYPE(CRLF</td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td>AMTHD(FSYS)</td>
<td></td>
</tr>
<tr>
<td>REWRITE</td>
<td>BUFFERED</td>
<td>CONSECUTIVE</td>
<td>AMTHD(FSYS)</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td>REGIONAL(1)</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>RECSIZE(n)</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>ENVIROMENT</td>
<td>VARYING</td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>INPUT UPDATE</td>
<td></td>
<td>TERMLBUF(n)</td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td>SCALARVARYING</td>
<td>TYPE(CRLF</td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td>AMTHD(FSYS)</td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>BUFFERED</td>
<td>REGIONAL(1)</td>
<td>AMTHD(FSYS)</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td></td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>RECSIZE(n)</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>ENVIROMENT</td>
<td>VARYING</td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>INPUT UPDATE</td>
<td></td>
<td>TERMLBUF(n)</td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td>SCALARVARYING</td>
<td>TYPE(CRLF</td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td>AMTHD(FSYS)</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. When creating a new data set
2. When printer-destined PL/I file
3. When associated with a PM terminal
4. When data set was not created by PL/I program
5. DIRECT applicable only to REGIONAL(1)
6. For REGIONAL(1)
7. Not applicable to REGIONAL(1)

**Table 10. Statements, attributes, and options for workstation VSAM data sets**

<table>
<thead>
<tr>
<th>Statements</th>
<th>File attributes</th>
<th>ENVIRONMENT options</th>
<th>DD_DDNAME options</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>ENVIRONMENT</td>
<td>ORGANIZATION(CONSECUTIVE)</td>
<td>AMTHD(DDM</td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td>GRAPHIC</td>
<td>APPEND(Y</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>RECSIZE(n)</td>
<td>ASA(Y</td>
</tr>
<tr>
<td></td>
<td>PRINT</td>
<td></td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>STREAM</td>
<td></td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td>GET</td>
<td>ENVIRONMENT</td>
<td>ORGANIZATION(CONSECUTIVE)</td>
<td>AMTHD(DDM</td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td>GRAPHIC</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>STREAM</td>
<td>RECSIZE(n)</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td>WRITE</td>
<td>BUFFERED</td>
<td>ORGANIZATION</td>
<td>AMTHD(DDM</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td>VSAM</td>
<td>ASA(Y</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>CTLASA</td>
<td>APPEND(Y</td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td>SCALARVARYING</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>ENVIRONMENT</td>
<td></td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td></td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUT UPDATE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 10. Statements, attributes, and options for workstation VSAM data sets (continued)

<table>
<thead>
<tr>
<th>Statements</th>
<th>File attributes</th>
<th>ENVIRONMENT options</th>
<th>DD_DDNAME options</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE</td>
<td>BUFFERED</td>
<td>ORGANIZATION</td>
<td>VSAM</td>
</tr>
<tr>
<td></td>
<td>ENVIRONMENT</td>
<td>CTLASA</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td>RECSIZE(n)</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td>SCALARVARYING</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td></td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>BUFFERED</td>
<td>ORGANIZATION</td>
<td>VSAM</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td>RECSIZE(n)</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>SCALARVARYING</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td></td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>ENVIRONMENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INPUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REWRITE</td>
<td>BUFFERED</td>
<td>ORGANIZATION</td>
<td>VSAM</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td>RECSIZE(n)</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>SCALARVARYING</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td></td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>ENVIRONMENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>BUFFERED</td>
<td>ORGANIZATION</td>
<td>VSAM</td>
</tr>
<tr>
<td></td>
<td>UNBUFFERED</td>
<td>RECSIZE(n)</td>
<td>file_spec</td>
</tr>
<tr>
<td></td>
<td>DIRECT</td>
<td>SCALARVARYING</td>
<td>RECSIZE(n)</td>
</tr>
<tr>
<td></td>
<td>SEQUENTIAL</td>
<td></td>
<td>SHARE(NONE</td>
</tr>
<tr>
<td></td>
<td>ENVIRONMENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FILE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KEYED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RECORD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. When creating a new data set
2. When printer-destined PL/I file
3. Does not apply to VSAM data sets

### DISPLAY statement input and output

The REPLY in DISPLAY is read from stdin. Output from the DISPLAY statement is directed to stdout by default. The syntax of the IBM.DISPLAY environment variable is:

```
IBM.DISPLAY= [std] [con]
```

- **std**
  - Specifies that the DISPLAY statement is to be associated with the standard output device. This is the default.

- **con**
  - Specifies that the DISPLAY statement is to be associated with the CON device.

You can redirect display statements to a file, for example:
DISPLAY I/O

```
set ibm.display=std

Hello: proc options(main);
    display('Hello!');
end;
```

After compiling and linking the program, you could invoke it from the command line by entering:
```
hello > hello1.out
```

The greater than sign redirects the output to the file that is specified after it, in this case HELLO1.OUT. This means that the word 'HELLO' is written in the file HELLO1.OUT.

### PL/I standard files (SYSPRINT and SYSIN)

SYSPRINT 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.

### Redirecting standard input, output, and error devices

You can also redirect standard input, standard output, and standard error devices to a file. You could use redirection in the following program, but you would first need to issue two SET DD: statements to allow the redirection to work. They are:
```
set dd:sysprint=stdout:
set dd:sysin=stdin:
```

```
Hello: proc options(main);
    put list('Hello!');
end;
```

After compiling and linking the program, you could invoke it from the command line by entering:
```
hello2 >
hello > hello2.out
```

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 since the PRINT attribute is applied to SYSPRINT by default.

READ statements can access data from stdin, however, they must specify an LRECL equal to 1.
Chapter 11. Defining and using consecutive data sets

The sections that follow describe consecutive data set organization and explain how to create, access, and update consecutive data sets.

In a data set with consecutive organization, records are organized solely on the basis of their successive physical positions. In other words, 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.

The information in this chapter applies to files using the CONSECUTIVE option of the ENVIRONMENT attribute that are associated with either a native or workstation VSAM data set. PL/I Presentation Manager supports only native data sets.

Printer-destined files

Printer-destined files are PL/I files with the PRINT attribute and record files declared with the CTLASA option of the ENVIRONMENT attribute. You can either print these files at your workstation or upload them to your mainframe.

The first character of each record is an American National Standard Institute (ANSI) carriage control character (see Table 11).

For STREAM files, PL/I inserts the character, based on the SKIP, LINE, or PAGE option (or control format item) of the PUT statement. For RECORD files with CTLASA, your program must insert the control characters in the first byte of each record.

If you want to print the data set from your workstation, select the ASA(N) option (it is the default). To keep the format for printing at the mainframe, select ASA(Y), which causes the control characters to be left untranslated.

Table 11. ANSI print control characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>Skip 1 line before printing</td>
</tr>
<tr>
<td>0</td>
<td>Skip 2 lines before printing</td>
</tr>
<tr>
<td>hyphen (-)</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>+</td>
<td>Do not skip any lines before printing</td>
</tr>
<tr>
<td>1</td>
<td>Skip to next page before printing</td>
</tr>
<tr>
<td>2</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>3</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>4</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>5</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>6</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>7</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>8</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>9</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>A</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>B</td>
<td>Skip 3 lines before printing</td>
</tr>
<tr>
<td>C</td>
<td>Skip 3 lines before printing</td>
</tr>
</tbody>
</table>

The translation to IBM Proprinter control characters is as follows:
Printer-destined files

### Table 12. IBM Proprinter equivalents to ANSI control characters

<table>
<thead>
<tr>
<th>ANSI Character</th>
<th>Proprinter Characters (in hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>0A</td>
</tr>
<tr>
<td>0</td>
<td>0A 0A</td>
</tr>
<tr>
<td>-</td>
<td>0A 0A 0A</td>
</tr>
<tr>
<td>+</td>
<td>0D</td>
</tr>
<tr>
<td>1</td>
<td>0C</td>
</tr>
<tr>
<td>2 to 9, A to C</td>
<td>0A 0A 0A</td>
</tr>
</tbody>
</table>

**Note:**

- Where:
  - 0A = Line feed
  - 0C = Form feed
  - 0D = Carriage return

Only the first five characters listed are translated by PL/I; the others are treated as hyphens (-).

### 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 you use in the DD:ddname environment variable for creating and accessing these data sets are summarized, and several examples of PL/I programs are included.

Data sets with the STREAM attribute are processed by stream-oriented data transmission. The stream-oriented data transmission allows your PL/I program to ignore block and record boundaries and to treat a data set as a continuous stream of data values. Data values are in either SBCS or DBCS format. You create and access data sets for stream-oriented data transmission 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 format 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 define files for stream-oriented data transmission by a file declaration with the following attributes:

```pli
declare file stream
  Filename file stream
    input [output [print]]
    environment(options);
```

Printer-destined files
Stream-oriented transmission

The FILE attribute is described in the PL/I Language Reference. The PRINT attribute is described further in “Using PRINT files” on page 261.

ENVIRONMENT options for stream-oriented data transmission

The ENVIRONMENT options you can use with stream-oriented data transmission are:
- CONSECUTIVE
- RECSIZE
- GRAPHIC
- ORGANIZATION(CONSECUTIVE).

You can find a description of these options and of their syntax in “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 233.

Creating a data set with stream I/O

To create a data set, use one of the following:
- ENVIRONMENT attribute
- DD:ddname environment variable
- TITLE option of the OPEN statement

Refer to “Using the TITLE option of the OPEN statement” on page 249 for more information on the TITLE option.

Essential information

When your application creates a STREAM file, it must supply a line size value for that file from one of the following sources:
- 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

The PL/I default is used when you do not supply any value. If you choose the LINESIZE option, it overrides all other sources. The RECSIZE option of the ENVIRONMENT attribute overrides the other RECSIZE options. RECSIZE specified in the TITLE option of the OPEN statement has precedence over the RECSIZE option of the DD:ddname environment variable.

If LINESIZE is not supplied, but a RECSIZE value is, PL/I derives line size value from RECSIZE as follows:
- A PRINT file with the ASA(N) option applied has a RECSIZE value of 4
- A PRINT file with the ASA(Y) option applied has the RECSIZE value of 1
- Otherwise, the value of RECSIZE is assigned to the line size value.

PL/I determines a default line size value based on attributes of the file and the type of associated data set. In cases where 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 when:
- The file has the PRINT attribute. In this case, the value is obtained from the tab control table (see Figure 11 on page 264).
- The associated data set is the terminal (CON:, STDOUT:, or STDERR:). In this case the value is 120.

PL/I always derives the record length of the data set from the line size value. A record length value is derived from the line size value as follows:
Stream-oriented transmission

- For a PRINT file, with the ASA(N) option applied, the value is line size + 4
- For a PRINT file, with the ASA(Y) option applied, the value is line size + 1
- Otherwise, the line size value is assigned to the record length value.

**Example**
Figure 8 on page 259 shows the use of stream-oriented data transmission to create a consecutive data set. The data is first read from the data set bday.inp that contains a list of names and birthdays of several people. Then a consecutive data set bday.oct is written that contains the names and birthdays of people whose birthdays are in October.

The command SET DD:SYSIN=BDAY.INP should be used to associate the disk file bday.inp with the input data set. If this file was not created by a PL/I program, the RECSIZE option must also be specified.

The command SET DD:WORK=BDAY.OCT should be used to associate the consecutive output file WORK with the disk data set bday.oct.
Accessing a data set with stream I/O

It is not necessary that a data set accessed using stream-oriented data transmission was created by stream-oriented data transmission. However, it must have CONSECUTIVE organization, and all the data in it must be in character or graphic
Stream-oriented transmission

form. You can open the associated file for input, and read the items the data set contains; or you can open the file for output, and extend the data set by adding items at the end.

To access a data set, you must use one of the following to identify it:
• ENVIRONMENT attribute
• DD:ddname environment variable
• TITLE option of the OPEN statement

Essential information
When your application accesses an existing STREAM file, PL/I must obtain a record length value for that file. 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
• An extended attribute of the data set
• 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 257.

PL/I uses a default record length value for an INPUT file when:
• The file is SYSIN, value = 80
• The file is associated with the terminal (CON:, SCREEN$: , STDOUT:, or STDERR: ), value = 120.

Example
The program in [Figure 9 on page 261] reads the data created by the program in [Figure 8 on page 259] and uses the data set SYSPRINT to display that data. The SYSPRINT data set is associated with the CON device, so if no dissociation is made prior to executing the program, the output is displayed on the screen. (For details on SYSPRINT, see “Using SYSIN and SYSPRINT files” on page 265.)
Using PRINT files

In a PL/I program, using a PRINT file provides a convenient means of controlling the layout of printed output from stream-oriented data transmission. PL/I 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.

PL/I 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 (see “Printer-destined files” on page 255).
Stream-oriented transmission

PL/I handles the PAGE, SKIP, and LINE options or format items by inserting the appropriate control character in the records. If the SKIP or the LINE option specifies more than a 3-line space, PL/I inserts sufficient blank records with appropriate control characters to accomplish the required spacing.

If a PRINT file is being transmitted to a terminal device, the PAGE, SKIP, and LINE options 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 by either:

• Specifying record length in your PL/I program using the RECSIZE option of the ENVIRONMENT attribute.
• Specifying line size in an OPEN statement using the LINESIZE option.
• Specifying record length in the TITLE option of the OPEN statement using the RECSIZE option.

RECSIZE must include the extra byte for the print control character; it must be 1 byte larger than the length of the printed line. LINESIZE refers to the number of characters in the printed line; PL/I adds the print control character.

Do not vary the line size for a file during execution by closing the file and opening it again with a new line size.

Since PRINT files have a default line size of 120 characters, you need not give any record length information for them.

Example:

Figure 10 on page 263 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.

Stream-oriented transmission

Figure 10. Creating a print file via stream data transmission. (The example in Figure 15 on page 275 prints this file)

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.
Stream-oriented transmission

The program in Figure 15 on page 275 uses record-oriented data transmission to print the table created by the program in Figure 10 on page 263.

Overriding the tab control table

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions, which are defined in the PL/I-defined tab control table. The tab control table is an external structure named PLITABS. Figure 11 shows its declaration.

```
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 11. Declaration of PLITABS. (Gives standard page size, line size and tabulating positions)

The definitions of the fields in the table are as follows:

**Offset**

Binary integer that gives the offset of Number_of_tabs, the field that indicates the number of tabs to be used, from the top of PLITABS.

**Pagesize**

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**

Binary integer that defines the default line size.

**Pagelength**

Binary integer that defines the default page length for printing at a terminal.

The value 0 indicates unformatted output.

**Fill1, Fill2, Fill3**

Three binary integers; reserved for future use.

**Number_of_tabs**

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-Tab5**

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 linker to resolve an external reference to PLITABS. You do this by including a PL/I
structure with the name PLITABS and the attributes EXTERNAL STATIC in the source program containing your main routine.

An example of the PL/I structure is shown in Figure 12. 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; FILL1, FILL2, and FILL3 can be omitted by adjusting the offset value by –6.

```pli
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 12. PL/I structure PLITABS for modifying the preset tab settings

Using SYSIN and SYSPRINT files

If you code GET or PUT statements without the FILE option, PL/I contextually assumes file SYSIN and SYSPRINT, respectively.

If you do not declare SYSPRINT, PL/I gives the file the attribute PRINT in addition to the normal default attributes; the complete set of attributes is:

```pli
  file stream print external
```

Since SYSPRINT is a PRINT file, a default line size of 120 characters is applied when the file is opened.

You can override the attributes given to SYSPRINT by PL/I by explicitly declaring or opening the file. However, when SYSPRINT is declared or opened as a STREAM OUTPUT file, the PRINT attribute is applied by default unless the INTERNAL attribute is also declared.

PL/I does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes.

Controlling input from the terminal

To enter data for an input file, do both of the following:

- Declare the input file explicitly or implicitly with the CONSECUTIVE environment option (all stream files meet this condition).
- 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.
Controlling input from the terminal

You can be prompted for input to stream files by a colon (:) if you specify PROMPT(Y), see “PROMPT” on page 243. You can 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.

If you do not specify PROMPT(Y), the default is to have no colon visible at the beginning of the line.

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. The hyphen is an explicit continuation character.

If your program includes output statements that prompt for input, 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:

```pli
put skip list('Enter next item:');
```

To inhibit the system prompt for the next GET statement, your own prompt must meet the following conditions:

- It must be either list-directed or edit-directed, and if list-directed, must be to a PRINT file.
- The file transmitting the prompt must be allocated to the terminal. If you are using the COPY option to copy the file at the terminal, the system prompt is not inhibited.

Using files conversationally

To have your programs interact with a user conversationally, use the terminal as an input and output device for consecutive files in the program. Any stream file can be used conversationally, because conversational I/O needs no special PL/I code.

Format of data

The data you enter on the terminal should have exactly the same format as stream input data in batch mode, except for the following 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; PL/I inserts a comma at the end of each line.

As an example, consider the following statement:

```pli
get list(I,J,K);
```

You could give the following response pressing the ENTER key after each item. (The colons only appear if you specify PROMPT(Y).

```
: 1
: 2
: 3
```

Entering the data on separate lines is equivalent to specifying:

```
: 1,2,3
```
Controlling input from the terminal

If you wish to continue an item on another line, you must end the first line with a continuation character (the hyphen). Otherwise, for a GET LIST or GET DATA statement, a comma is inserted. For a GET EDIT statement, the item is padded.

- **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 is padded to the correct length.
  
  Consider the following PL/I statement:
  
  ```pli
  get edit(Name)(a(15));
  ```
  
  You could enter these five characters followed immediately by the ENTER.

  ```pli
  SMITH
  ```

  The item is padded with 10 blanks, so that the program receives a string 15 characters long. If you wish 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. Otherwise, the first line transmitted would be padded and treated as the complete data item.

- **SKIP option or format item:** A SKIP in a GET statement ignores the data not yet entered. All uses of SKIP(n) where \( n \) is greater than one are taken to mean SKIP(1). SKIP(1) is taken to mean 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 is not necessarily 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.

Capital and lowercase letters

For both stream and record files, character strings are transmitted to the program as entered in lowercase or 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 indication and raise the ENDFILE condition.

Note that the sequence of characters /* in a line is not an end-of-file indication when input is attached to a file instead of a terminal. For example:

```pli
   cat input.txt | ./plipgm
```

or

```pli
   ./plipgm < input.txt
```

where a line containing only /* from the input.txt file is not an end-of-file indication.

When using a terminal, use the control-D key sequence to generate an end-of-file indication.
Controlling output to the terminal

At your screen, you can display data from a PL/I file that has been both:

- Declared explicitly or implicitly with the CONSECUTIVE environment option. All stream files meet this condition.
- Allocated to the terminal device (CON:, STDOUT:, SCREEN$, or STDERR:).

The standard output file SYSPRINT generally meets both these conditions.

Format of PRINT files

Data from SYSPRINT or other PRINT files is not normally formatted into columns and 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 carriage return.

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 264. For other than standard layout, use the information about PLITABS provided in Figure 11 on page 264. You can also use PLITABS to alter the tabulating positions of list-directed and data-directed output.

Tabulating of list-directed and data-directed output is achieved by transmission of blank (space) characters.

Stream and record files

You can allocate both stream and record files to the terminal. However, if you allocate more than one file to the terminal and one or more is a record file, the file output 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 input and output statements are executed.

For stream and record files, characters are displayed on the terminal as they are held in the program. Both capital and lowercase characters can be displayed.

Example of an interactive program

The example program in Figure 13 on page 269 creates a consecutive data set PHONES using a dialog with the user. By default, SYSIN is associated with the CON device. You can override this association by setting an environment variable for the SYSIN file or by using the TITLE option on the OPEN statement. The output data set is associated with a disk file INT1.DAT and contains names and phone numbers that the user enters from the keyboard.
Using record-oriented I/O

INT1: proc options(main);

dcl Phones stream env(recsize(40));

dcl Eof bit(1) init('0'b);

dcl 1 PhoneBookEntry,

            3 NameField char(19),

            3 PhoneNumber char(21);

dcl InArea char(40);

open file (Phones) output;

on endfile(sysin) Eof='1'b;

/* start creating phone book */
put list('Please enter name:');
get edit(NameField)(a(19));
if ¬Eof then
do;
put list('Please enter number:');
get edit(PhoneNumber)(a(21));
end;
do while ¬Eof;
put file(Phones) edit(PhoneBookEntry)(a(40));
put list('Please enter name:');
get edit(NameField)(a(19));
if ¬Eof then
do;
put list('Please enter number:');
get edit(PhoneNumber)(a(21));
end;
end;

close file(Phones);

end INT1;

Figure 13. A sample interactive program

Using record-oriented I/O

PL/I supports various types of data sets with the RECORD attribute. This section covers how to use record-oriented I/O with consecutive data sets.

Table 13 on page 270 lists the data transmission statements and options that you can use to create and access a consecutive data set using record-oriented I/O.

Chapter 11. Defining and using consecutive data sets
Using record-oriented I/O

A CONSECUTIVE file that is associated with a DDM direct or keyed data set can be opened only for INPUT. PL/I raises UNDEFINEDFILE if an attempt is made to open such a file for OUTPUT or UPDATE.

Table 13. Statements and options allowed for creating and accessing consecutive data sets

<table>
<thead>
<tr>
<th>File Declaration</th>
<th>Valid Statements, with Options You Must Specify</th>
<th>Other Options you can Specify</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL OUTPUT BUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td>SET(pointer reference)</td>
</tr>
<tr>
<td></td>
<td>LOCATE based-variable FILE(file-reference);</td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL OUTPUT UNBUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL INPUT BUFFERED</td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) IGNORE(expression);</td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL INPUT UNBUFFERED</td>
<td>READ FILE(file-reference) INPUT(reference);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) IGNORE(expression);</td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL UPDATE BUFFERED</td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) IGNORE(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference) FROM(reference);</td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL UPDATE UNBUFFERED</td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td>FROM(reference)</td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) IGNORE(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference);</td>
<td></td>
</tr>
</tbody>
</table>

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);.

Defining files using record I/O

You define files for record-oriented data transmission by using a file declaration with the following attributes:
Using record-oriented I/O

declare
    filename file record
        input | output | update
        sequential
        buffered | unbuffered
        environment(options);

The file attributes are described in the PL/I Language Reference.

ENVIRONMENT options for record-oriented data transmission
The ENVIRONMENT options applicable to consecutive data sets for record-oriented data transmission are:
- CONSECUTIVE
- CTLASA
- ORGANIZATION(CONSECUTIVE)
- RECSIZE
- SCALARVARYING

You can find a description of these options and of their syntax in “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 233.

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 13 on page 270 shows the statements and options for creating a consecutive data set.

To create a data set, you must give PL/I certain information either in the ENVIRONMENT attribute, in a DD:ddname environment variable, or in the TITLE option of the OPEN statement.

Essential information
When you create a consecutive data set you must specify:
- 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 (see “Attempting to use files not associated with data sets” on page 250).
- The record length. You can specify the record length 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 (CON:, 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
Once 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. For an example of a program that accesses and updates a consecutive data set, see Figure 14 on page 273.

If you open the file for output, and wish to extend the data set by adding records at the end, you need not specify APPEND(Y) in the DD:ddname environment variable, since this is the default. If you specify APPEND(N), the data set is overwritten. If you open a file for updating, you can only update records in their existing sequence, and if you want to insert records, you must create a new data set. You cannot change the record length of an existing data set.
Using record-oriented I/O

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. Every record that is retrieved, however, need not be rewritten. A REWRITE statement always updates the last record read.

Consider the following:
```
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.

To access a data set, you must identify it to PL/I using the TITLE option of the OPEN statement or a DD:ddname environment variable.

Table 13 on page 270 shows the statements and options for accessing and updating a consecutive data set.

Essential information

When your application accesses an existing RECORD file, PL/I must obtain a record length value for that file. The value can come from one of the following sources:
- 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.

PL/I uses a default record length value for an INPUT file when:
- The file is SYSIN. In this case, the value used is 80.
- The file is associated with the terminal. In this case, the value used is 120.

Examples of consecutive data sets

Creating and accessing consecutive data sets are illustrated in the program in Figure 14 on page 273. The program merges the contents of two PL/I files INPUT1 and INPUT2, and writes them onto a new PL/I file, OUT. INPUT1 and INPUT2 are associated with the disk files EVENS.INP and ODDS.INP, respectively, and contain 6-byte records arranged in ASCII collating sequence.
Using record-oriented I/O

```
MERGE: proc options(main);

   put skip list('START CON4 TEST');

   dcl Input1 file record sequential input env(recsize(6));
   dcl Input2 file record sequential input env(recsize(6));
   dcl Out file record sequential env(recsize(15));
   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(6) based(a); /* item from Input1 */
   dcl Item2 char(6) based(b); /* item from Input2 */
   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),
       file(Input2),
       file(Out) output;

   read file(Input1) set(A); /* priming read */
   read file(Input2) set(B);
```

*Figure 14. Merge Sort—Creating and accessing a consecutive data set*
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;
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(Item1); /* display Out file */
do while (Out_Eof = False);
  put file(Sysprint) skip edit(Item1) (a);
  read file(Out) into(Item1);
end;
close file(Out);
put skip list('END CON4 TEST');
end MERGE;

Here is a sample of evens.inp:

BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ

Here is a sample of odds.inp:

AAAAAA
CCCCCC
EEEEEEE
GGGGGG
IIIIII
KKKKKK

Merge Sort—Creating and accessing a consecutive data set

The program in Figure 15 on page 275 uses record-oriented data transmission to write the table created by the program in Figure 10 on page 263 to a file called
Using record-oriented I/O

out.txt.

PRT: proc options(main);
   put skip list('START CON5 TEST');
   dcl Table file record input sequential;
   dcl Output file record output sequential env(recsize(200));
   dcl Line char(102) 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(Output);
   read file(Table) into(Line); /* priming read */
   do while (Table_Eof = False);
      if Line='' then /* insert blank lines */
         Line= ' ';
      write file(Output) from(Line);
      read file(Table) into(Line);
   end;
   close file(Table),
      file(Output);
   put skip list('END CON5 TEST');
end PRT;

Figure 15. Printing record-oriented data transmission
Using record-oriented I/O
Chapter 12. Defining and using regional data sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. Creating and accessing regional data sets are also discussed.

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. The regions are numbered in succession, beginning with zero, and a record can be accessed by specifying its region number 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 data access time. This type of optimization is not available with consecutive organization, in which successive records are written in strict physical sequence.

You can create a regional data set in a manner similar to a consecutive 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. Once 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.

PL/I supports REGIONAL(1) data sets. See Table 14 for a list of the data transmission statements and options that you can use to create and access a REGIONAL(1) data set.

Table 14. Statements and options allowed for creating and accessing regional data sets

<table>
<thead>
<tr>
<th>File Declaration</th>
<th>Valid Statements, With Options You Must Include</th>
<th>Other Options You Can Also Include</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL OUTPUT BUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); LOCATE based-variable FROM(file-reference) KEYFROM(expression);</td>
<td>SET(pointer-reference)</td>
</tr>
<tr>
<td>SEQUENTIAL OUTPUT UNBUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
</tbody>
</table>
### Regional data sets

<table>
<thead>
<tr>
<th>File Declaration</th>
<th>Valid Statements, With Options You Must Include</th>
<th>Other Options You Can Also Include</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SEQUENTIAL INPUT</strong></td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td>KEYTO(reference)</td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference);</td>
<td>KEYTO(reference)</td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) IGNORE(expression);</td>
<td>KEYTO(reference)</td>
</tr>
<tr>
<td><strong>SEQUENTIAL UPDATE</strong></td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td>KEYTO(reference)</td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference);</td>
<td>KEYTO(reference)</td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) IGNORE(expression);</td>
<td>KEYTO(reference)</td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference);</td>
<td>FROM(reference)</td>
</tr>
<tr>
<td><strong>DIRECT OUTPUT</strong></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECT INPUT</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECT UPDATE</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference) FROM(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE FILE(file-reference) KEY(expression);</td>
<td></td>
</tr>
</tbody>
</table>
### Regional data sets

**Table 14. Statements and options allowed for creating and accessing regional data sets (continued)**

<table>
<thead>
<tr>
<th>File Declaration¹</th>
<th>Valid Statements,² With Options You Must Include</th>
<th>Other Options You Can Also Include</th>
</tr>
</thead>
</table>

**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 The file cannot have the UPDATE attribute when creating new data sets.

---

### Defining files for a regional data set

Use a file declaration with the following attributes to define a sequential regional data set:

```pli
declare
  filename file record
  input | output | update
  sequential
  buffered | unbuffered
  [keyed]
  environment(options);
```

To define a direct regional data set, use a file declaration with the following attributes:

```pli
declare
  filename file record
  input | output | update
  direct
  unbuffered
  [keyed]
  environment(options);
```

File attributes are described in the *PL/I Language Reference*.

### Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to regional data sets are:

- FIXED
- REGIONAL(1)
- RECSIZE
- SCALARVARYING

These options are described in “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 233.

### Essential information for creating and accessing regional data sets

To create a regional data set, you must give PL/I certain information, either in the ENVIRONMENT attribute or in the DD:ddname environment variable.

You must supply the following information when creating a regional data set:
Defining files for a regional data set

- The name of the data set associated with your PL/I file. A data set with REGIONAL(1) organization can exist only on a direct-access storage device (see "Attempting to use files not associated with data sets" on page 250).
- The record length. You can specify the record length using the RECSIZE option of the ENVIRONMENT attribute or of the DD:ddname environment variable or in the TITLE option of the OPEN statement.
- The extent (the number of regions) of the data set. You specify this with the RECCOUNT option of the DD:ddname environment variable. The default for RECCOUNT is 50.

Using keys with regional data sets

Source keys are used to access REGIONAL(1) data sets. 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 is the region number.

Using REGIONAL(1) data sets

In a REGIONAL(1) data set, 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 2147483647. If the region number exceeds this figure, it is treated as modulo 2147483648; for instance, 2147483658 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 region number; the first embedded blank, if any, terminates the region number. If more than 10 characters appear in the source key, only the rightmost 10 are used as the region number; if there are fewer than 10 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 'FFX' 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.

Creating a REGIONAL(1) data set

You can create a REGIONAL(1) data set either sequentially or by direct-access. Table 14 on page 277 shows the statements and options for creating a regional data set.

When you create the data set, opening the file causes the data set to be filled with dummy records. You must present records in ascending order of region numbers for a SEQUENTIAL OUTPUT file. If there is an error in the sequence, or if you present a duplicate key, the KEY condition is raised. If you use a DIRECT OUTPUT file to create the data set, you can present records in random order. If you present a duplicate region number, the existing record is overwritten.
Using REGIONAL(1) data sets

If you create a data set using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement might raise the ERROR condition.

Example

Creating a REGIONAL(1) data set is illustrated in Figure 16. The data set is a list of telephone extensions with the names of the subscribers to whom they are allocated. The telephone extensions correspond with the region numbers in the data set; the data in each occupied region being a subscriber's name.

```plaintext
/********************************************************************/
/* */
/* DESCRIPTION */
/* Create a REGIONAL(1) data set. */
/* */
/* USAGE */
/* The following commands are required to establish */
/* the environment variables to run this program: */
/* */
/* SET DD:SYSIN=CRG.INP,RECSIZE(30) */
/* */
/* SET DD:NOS=NOS.DAT,RECCOUNT(100) */
/* */
/* */
/************************************************************************/

CRR1: proc options(main);
  dcl Nos file record output direct keyed
       env(regional(1) recsize(20));
  dcl Sysin file input record;
  dcl 1 In_Area,
       2 Name char(20),
       2 Number char(2);
  dcl IoField char(20);
  dcl Sysin_EOF bit (1) init('0'b);
  dcl Ntemp fixed(15);
  on endfile (Sysin) Sysin_EOF = '1'b;
  open file(Nos);
  read file(Sysin) into(In_Area);
  do while(¬Sysin_EOF);
    IoField = Name;
    Ntemp = Number;
    write file(Nos) from(IoField) keyfrom(Ntemp);
    put file(sysprint) skip edit (In_Area) (a);
    read file(Sysin) into(In_Area);
  end;
  close file(Nos);
end CRR1;
```

Figure 16. Creating a REGIONAL(1) data set

Chapter 12. Defining and using regional data sets  281
The execution time input file, crg.inp, might look like this:

```
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.O.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
```

Creating a REGIONAL(1) data set (continued)

Accessing and updating a REGIONAL(1) data set

Once 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 14 on page 277 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, since you can use the KEYTO option. If the target character string referenced in the KEYTO option has more than 10 characters, the value returned (the 10-character region number) is padded on the left with blanks. If the target string has fewer than 10 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. A discussion of using READ and REWRITE statements can be found in "Accessing and updating a data set with record I/O" on page 271.
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 turned into a dummy record.

Replacement
The record you specify by the source key in a REWRITE statement, whether dummy or actual, is replaced.

Example

Updating a REGIONAL(1) data set is illustrated in Figure 17 on page 284. This program 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.
Using REGIONAL(1) data sets

/* DESCRIPTION */
/* Update a REGIONAL(1) data set. */
/* */
/* USAGE */
/* The following commands are required to establish */
/* the environment variables to run this program: */
/* */
/* SET DD:SYSIN=ACR.INP,RECSIZE(30) */
/* SET DD:NOS=NOS.DAT,APPEND(Y) */
/* */
/* Note: This sample program is using the regional data set, */
/* nos.dat, created by the previous sample program CRR1. */
/* */
/***********************************************************/

ACR1: proc options(main);

dcl Nos file record keyed env(regional(1));
dcl Sysin file input record;
dcl Sysin_EOF bit (1) init('0'b);
dcl Nos_EOF bit (1) init('0'b);
dcl 1 In_Area,
  2 Name char(20),
  2 (CNewNo,COldNo) char( 2),
  2 In_Area_1 char( 1),
  2 Code char( 1);
dcl IoField char(20);
dcl Byte char(1) def IoField;
dcl NewNo fixed(15);
dcl OldNo fixed(15);

  on endfile (Sysin) Sysin_EOF = '1'b;
  open file (Nos) direct update;
  read file(Sysin) into(In_Area);

Figure 17. Updating a REGIONAL(1) data set
do while(~Sysin_Eof);
  if CNewNo ¬=' ' then
    NewNo = CNewNo;
  else
    NewNo = 0;
  if COldNo ¬=' ' then
    OldNo = COldNo;
  else
    OldNo = 0;
  select(Code);
  when('A','C')
  do;
    if Code = 'C' then
      delete file(Nos) key(OldNo);
      read file(Nos) key(NewNo) into(IoField);
      /* we must test to see if the record exists */
      /* if it doesn't exist we create a record there */
      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(In_Area);
end ACR1;

At execution time, the input file, ACR.INP, could look like this:

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 X
ROLF,D.F. 14 D
ELLIOTT,D. 4285 C
HASTINGS,G.M. 31 D
BRAMLEY,O.H. 4928 C
Using REGIONAL(1) data sets
Chapter 13. Defining and using workstation VSAM data sets

This chapter describes how you use Virtual Storage Access Method (VSAM) data sets on your workstation—including Distributed Data Management (DDM), ISAM, and BTRIEVE data sets—for record-oriented data transmission.

Platform distinction

Three access methods are discussed in connection with the PL/I workstation products; however, not all three methods are supported on every platform. Use the following as a guideline:

- DDM—supported on AIX only
- ISAM—supported on AIX and Windows
- BTRIEVE—supported on Windows only

This chapter also describes the statements you use to access the three types of VSAM data sets—sequential, keyed, and direct. In many ways, workstation VSAM is similar to the VSAM on the mainframe. On the workstation, the terms sequential, keyed and direct are similar to the VSAM entry-sequenced data set, key-sequenced data set, and relative record data set.

The chapter concludes with a series of examples showing the PL/I statements and DD:ddname environment variables necessary to create and access workstation VSAM data sets.

Moving data between the workstation and mainframe

To convert mainframe VSAM files to the corresponding DDM, ISAM, or BTRIEVE files, follow the procedure documented in the prolog for the LODVSAM utility. Make sure you specify the appropriate access method AMTHD(DDM | ISAM | BTRIEVE).

To convert DDM, ISAM, or BTRIEVE files to corresponding mainframe VSAM files, follow the procedure documented in the prolog for the RELOAD utility. These utilities are supported on PL/I for Windows, but are not currently available on PL/I for AIX.

Workstation VSAM organization

PL/I supports workstation VSAM sequential, keyed, and direct data sets. These correspond to PL/I consecutive, indexed, and relative data set organizations, respectively.

Both sequential and keyed access are possible with all three types of data sets. With keyed data sets, the key, which is part of the logical record, is used for keyed access; keyed access is possible for direct data sets using relative record numbers. Keyed access is also possible for sequential data sets using the sequential record value as a key.

All workstation VSAM data sets are stored on direct-access storage devices. The physical organization of workstation VSAM data sets differs from those used by other access methods.
Creating and accessing workstation VSAM data sets

Your PL/I application can create workstation VSAM data sets, or it can access VSAM data sets created by other programs. When you open a file to be associated with a workstation VSAM data set, and that data set does not exist, PL/I creates it using the attributes and options you specify in the DECLARE statement or in a DD:ddname environment variable.

When your application accesses an existing VSAM data set, PL/I determines its type—sequential, direct, or keyed.

The operation of writing the initial data into a newly-created VSAM data set is referred to as **loading** in this publication.

Are you using the right access method?

Use each of the access methods, DDM | ISAM | BTRIEVE, to access data sets that were created with that particular access method. For example, you cannot use the ISAM access method to access data sets you created with the BTRIEVE access method.

Determining which type of workstation VSAM data set you need

Use the three different types of data sets according to the following purposes:

- **Use sequential data sets** for data that you access primarily in the order in which the records were created (or the reverse order).
- **Use keyed 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 direct 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).

Accessing records in workstation VSAM data sets

You can access records in all types of workstation VSAM data sets either directly by means of a key or sequentially (backward or forward). You can also use a combination of the two ways, in which you select a starting point with a key and then read forward or backward from that point.

Table 15 on page 289 shows how data could be stored in the three different types of workstation VSAM data sets and illustrates their respective advantages and disadvantages.
### Workstation VSAM organization

#### Table 15. Types and advantages of workstation VSAM data sets

<table>
<thead>
<tr>
<th>Data Set Type</th>
<th>Method of Loading</th>
<th>Method of Reading</th>
<th>Method of Updating</th>
<th>Pros and Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Sequentially</td>
<td>SEQUENTIAL</td>
<td>New records at end only</td>
<td>Advantages: Simple fast creation</td>
</tr>
<tr>
<td></td>
<td>(forward only)</td>
<td>backward or forward</td>
<td>Access can be sequential or KEYED</td>
<td>Uses: For uses where data is primarily accessed sequentially</td>
</tr>
<tr>
<td></td>
<td>The sequential record value of each record can be obtained and used as a key</td>
<td>Positioning by key followed by sequential either backward or forward</td>
<td>Record deletion allowed</td>
<td></td>
</tr>
<tr>
<td>Keyed</td>
<td>Either sequentially or randomly by key</td>
<td>KEYED by specifying key of record</td>
<td>KEYED specifying a key</td>
<td>Advantages: Complete access and updating</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SEQUENTIAL</td>
<td>SEQUENTIAL following positioning by key</td>
<td>Uses: For uses where access is related to key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>backward or forward in order of any index</td>
<td>Record deletion allowed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Positioning by key followed by sequential reading either backward or forward</td>
<td>Record insertion allowed</td>
<td></td>
</tr>
<tr>
<td>Direct</td>
<td>Sequentially</td>
<td>KEYED</td>
<td>Sequentially starting at a specified slot and continuing with next slot</td>
<td>Advantages: Speedy access to record by number</td>
</tr>
<tr>
<td></td>
<td>starting from slot 1</td>
<td>specifying numbers as key</td>
<td>Keyed specifying numbers as key</td>
<td>Disadvantages: Structure tied to numbering sequences</td>
</tr>
<tr>
<td></td>
<td>KEYED specifying number of slot</td>
<td>Sequential forward or backward omitting empty records</td>
<td>Record deletion allowed</td>
<td>Uses: For use where records are accessed by number</td>
</tr>
<tr>
<td></td>
<td>Positioning by key followed by sequential writes</td>
<td></td>
<td>Record insertion into empty slots allowed</td>
<td></td>
</tr>
</tbody>
</table>

#### Using keys for workstation VSAM data sets

All workstation VSAM data sets can have keys associated with their records. For keyed data sets, the key is a defined field within the logical record. For sequential data sets, the key is the sequential record value of the record. For relative record data sets, the key is a relative record number.

#### Using keys for workstation VSAM keyed data sets

Keys for keyed data sets 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.
The ways you can reference the keys in the KEY, KEYFROM, and KEYTO options are as described under “KEY(expression) Option,” “KEYFROM(expression) Option,” and “KEYTO(reference) Option” in the PL/I Language Reference.

**Using sequential record values**
Sequential record values allow you to use keyed access on a sequential data set associated with a KEYED SEQUENTIAL file.

**BTRIEVE and ISAM**
The sequential record values, or keys, are character strings of length 7, and their values are defined by workstation VSAM.

You cannot construct or manipulate sequential record values in PL/I; you can, however, compare their values in order to determine the relative positions of records within the data set. Sequential record values are not normally printable.

You can obtain the sequential record value 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 a sequential record value obtained in either of these ways in the KEY option of a READ or REWRITE statement.

**Using relative record numbers**
Records in a direct data set are identified by a relative record number that starts at 1 and is incremented 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 10. 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 10 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 10, with leading zeros suppressed.

**Choosing a data set type**
When planning your application, you must first decide which type of data set to use. There are three types of workstation VSAM data sets and five types of non-VSAM data sets available to you. Workstation VSAM data sets can provide all the functions of the other types of data sets, plus additional functions available only with workstation VSAM. Workstation VSAM can usually match, or even improve upon, the performance of other data set types. However, workstation VSAM is more subject to performance degradation through misuse of functions.

Table 15 on page 289 shows you the possibilities available with each type of workstation VSAM data set. When choosing between the workstation VSAM data set types, you should base your decision on the most common sequence in which your program accesses your data. You can use the following procedure to ensure a combination of data sets and indexes that provide the function you require:

1. Determine the type of data and how it is accessed.
   a. Primarily sequentially — favors ESDS.
   b. Primarily by key — favors KSDS.
   c. Primarily by number — favors RRDS.
2. Determine how you 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
Choosing a data set type

alternative for some applications. Table 16 on page 292 shows types of processing that you can use on alternate index paths.

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 has unique or nonunique keys. The use of nonunique keys can limit key processing. However, it might also be impractical to assume that you 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. Figure 18 shows the workstation VSAM data sets and the file attributes that you can specify.

Do not try to access a dummy workstation VSAM data set, because you will receive an error message indicating that you have an undefined file.

Table 17 on page 295, Table 18 on page 298, and Table 19 on page 305 show the statements allowed for sequential data sets, keyed data sets, and direct data sets, respectively.

<table>
<thead>
<tr>
<th></th>
<th>SEQUENTIAL</th>
<th>KEYED SEQUENTIAL</th>
<th>DIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>ESDS</td>
<td>ESDS</td>
<td>KSDS</td>
</tr>
<tr>
<td></td>
<td>KSDS</td>
<td>KSDS</td>
<td>RRDS</td>
</tr>
<tr>
<td></td>
<td>RRDS</td>
<td>RRDS</td>
<td>Path(U)</td>
</tr>
<tr>
<td></td>
<td>Path(N)</td>
<td>Path(N)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Path(U)</td>
<td>Path(U)</td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td>ESDS</td>
<td>ESDS</td>
<td>KSDS</td>
</tr>
<tr>
<td></td>
<td>RRDS</td>
<td>RRDS</td>
<td>RRDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Path(U)</td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>ESDS</td>
<td>ESDS</td>
<td>KSDS</td>
</tr>
<tr>
<td></td>
<td>RRDS</td>
<td>RRDS</td>
<td>RRDS</td>
</tr>
<tr>
<td></td>
<td>Path(N)</td>
<td>Path(N)</td>
<td>Path(U)</td>
</tr>
<tr>
<td></td>
<td>Path(U)</td>
<td>Path(U)</td>
<td></td>
</tr>
</tbody>
</table>

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

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.

Figure 18. VSAM data sets and allowed file attributes
### Choosing a data set type

#### Table 16. Processing allowed on alternate index paths

<table>
<thead>
<tr>
<th>Base Cluster Type</th>
<th>Alternate Index Key Type</th>
<th>Processing</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>KSDS</td>
<td>Unique key</td>
<td>As normal KSDS</td>
<td>May not modify key of access.</td>
</tr>
<tr>
<td></td>
<td>Nonunique key</td>
<td>Limited keyed access</td>
<td>May not modify key of access.</td>
</tr>
<tr>
<td>ESDS</td>
<td>Unique key</td>
<td>As KSDS</td>
<td>No deletion.</td>
</tr>
<tr>
<td></td>
<td>Nonunique key</td>
<td>Limited keyed access</td>
<td>May not modify key of access.</td>
</tr>
</tbody>
</table>

#### Defining files for workstation VSAM data sets

You define a workstation VSAM sequential data set by using a file declaration with the following attributes:

```pli
  dcl Filename file record
    input | output | update
    sequential
    buffered | [keyed]
    environment(options);
```

You define a workstation VSAM keyed data set by using a file declaration with the following attributes:

```pli
  dcl Filename file record
    input | output | update
    sequential | direct
    buffered | [keyed]
    environment(options);
```

You define a workstation VSAM direct data set by using a file declaration with the following attributes:

```pli
  dcl Filename file record
    input | output | update
    direct | sequential
    buffered | [keyed]
    environment(options);
```

The file attributes are described in the *PL/I Language Reference* for this product. Options of the `ENVIRONMENT` attribute are discussed below.

### Specifying options of the PL/I `ENVIRONMENT` attribute

Many of the options of the PL/I `ENVIRONMENT` attribute affecting data set structure are not needed for workstation VSAM data sets. If you specify them, they are either ignored or are 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.

The `ENVIRONMENT` options applicable to workstation VSAM data sets are:
For a complete explanation of these ENVIRONMENT options and how to use them, see “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 233. In addition to this list of ENVIRONMENT options, there is a set of options that can be used with a DD statement, see “Specifying characteristics using DD:ddname environment variables” on page 239.

Adapting existing programs for workstation VSAM

This section is intended primarily for PL/I for z/OS users who are transferring programs to the workstation.

In most cases, if your PL/I program uses files declared with ENVIRONMENT (CONSECUTIVE) or ENVIRONMENT(INDEXED) or with no PL/I ENVIRONMENT attribute, it can access workstation VSAM data sets without alteration. PL/I detects that a workstation VSAM data set is being opened and can provide the correct access.

You can readily adapt existing programs with CONSECUTIVE, INDEXED, REGIONAL(1) or VSAM files for use with workstation VSAM data sets. Programs with consecutive files might not need alteration, and there is never any necessity to alter programs with indexed files unless the logic depends on EXCLUSIVE files. Programs with REGIONAL(1) data sets require only minor revision.

The following sections tell you what modifications you might need to make in order to adapt files for the workstation.

Adapting programs using CONSECUTIVE files

There is no concept of fixed-length records in DDM, but there is in ISAM and BTRIEVE. If your program relies on the RECORD condition to detect incorrect length records, it does not function in the same way using workstation VSAM data sets as it does with non-workstation VSAM data sets.

If the logic of the program depends on raising the RECORD condition when a record of an incorrect length is found, you must write your own code to check for the record length and take the necessary action. This is because records of any length up to the maximum specified are allowed in workstation VSAM data sets.

Adapting programs using INDEXED files

Compatibility is provided for INDEXED files. For files that you declare with the INDEXED ENVIRONMENT option, PL/I associates the file with a workstation VSAM keyed data set. UNDEFINEDFILE is raised if the data set is any other type.

Because mainframe ISAM record handling differs in detail from workstation VSAM record handling, workstation VSAM processing might not always give the required result.
Defining files for workstation VSAM data sets

You should remove dependence on the RECORD condition, and insert your own code to check for record length if this is necessary. You should also remove any checking for deleted records.

Adapting programs using REGIONAL(1) files
You can alter programs using REGIONAL(1) data sets to use workstation VSAM direct data sets. Remove REGIONAL(1) and any other implementation-dependent options from the file declaration and replace them with ENV(ORGANIZATION(RELATIVE)). You should also remove any checking for deleted records, because workstation VSAM deleted records are not accessible to you.

Adapting programs using VSAM files
If you use the VSAM ENVIRONMENT option, the associated workstation VSAM data set must exist before the file is opened. You can create your data sets with a simple program. Figure 19 is an example of creating a workstation VSAM keyed data set.

```pli
/*****************************/
/*                        */
/* NAME - ISAM0.PL1        */
/*                        */
/* DESCRIPTION            */
/* Create an ISAM Keyed data set */
/*                        */
/*                        */
/*****************************/

NewVSAM: proc options(main);
  declare
    NewFile keyed record output file
      env(organization(indexed)
        recsize(80)
        keylength(8)
        keyloc(17)
      );
  open file(NewFile) title('/keynames.dat');
  close file(NewFile);
End NewVSAM;
```

Figure 19. Creating a workstation VSAM keyed data set

If the data set named KEYNAMES.DAT does not already exist, PL/I creates it with that name when the OPEN statement is executed.
### Using workstation VSAM sequential data sets

The statements and options allowed for files associated with a workstation VSAM sequential data set are shown in Table 17.

Table 17. Statements and options allowed for loading and accessing workstation VSAM sequential data sets

<table>
<thead>
<tr>
<th>File declaration</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL OUTPUT BUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference); LOCATE based-variable FILE(file-reference);</td>
<td>KEYTO(reference) SET(pointer-reference)</td>
</tr>
<tr>
<td>SEQUENTIAL OUTPUT UNBUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td>KEYTO(reference)</td>
</tr>
</tbody>
</table>
| SEQUENTIAL INPUT BUFFERED | READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference); | KEYTO(reference) or KEY(expression)
| SEQUENTIAL INPUT UNBUFFERED | READ FILE(file-reference) INTO(reference); READ FILE(file-reference); KEYTO(reference) or KEY(expression)
| SEQUENTIAL UPDATE BUFFERED | READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference); WRITE FILE(file-reference) FROM(reference); REWRITE FILE(file-reference); DELETE FILE(file-reference); | KEYTO(reference) and/or KEY(expression)
| SEQUENTIAL UPDATE UNBUFFERED | READ FILE(file-reference) INTO(reference); WRITE FILE(file-reference) FROM(reference); REWRITE FILE(file-reference) FROM(reference); | KEY(expression)

---

Chapter 13. Defining and using workstation VSAM data sets 295
Using workstation VSAM sequential data sets

Table 17. Statements and options allowed for loading and accessing workstation VSAM sequential data sets (continued)

<table>
<thead>
<tr>
<th>File declaration¹</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
</table>

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 sequential record value, previously obtained by means of the KEYTO option.

Using a sequential file to access a workstation VSAM sequential data set

When a sequential data set is being loaded, the associated file must be opened for SEQUENTIAL OUTPUT. The records are stored in the order in which they are presented.

You can use the KEYTO option to obtain the sequential record value of each record as it is written. You can subsequently use these keys to achieve keyed access to the data set.

You can open a SEQUENTIAL file that is used to access a workstation VSAM sequential data set 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 occurs 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 sequential record value of the records that are read. If you use the KEY option, the record that is recovered is the one with the sequential record value 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 sequential record value if you use the KEY option; otherwise, it is the record accessed on the previous READ.

Defining and loading a workstation VSAM sequential data set

[Figure 20 on page 297] is an example of a program that defines and loads a workstation VSAM sequential data set. You can also use this program to process a DDM or SFS file by modifying the AMTHD specification.

The PL/I program writes the data set using a SEQUENTIAL OUTPUT file and a WRITE FROM statement.

The sequential record values of the records could have been obtained during the writing for subsequent use as keys in a KEYED file. To do this, a suitable variable would have to be declared to hold the key and the WRITE...KEYTO statement used. For example:
Using workstation VSAM sequential data sets

```
dcl Chars char(7); /* DDM uses 4; BTRIEVE and ISAM use 7 as shown */
write file(Famfile) from (String)
   keyto(Chars);
dcl Chars char(4); /* DDM uses 4 */
write file(Famfile) from (String)
   keyto(Chars);
```

The keys would not normally be printable, but could be retained for subsequent use.

```
CREATE: proc options(main);
   put skip list('START ISAM1 TEST');

dcl
   FamFile file sequential output
      env(organization(consecutive)),
      In file record input,
      Eof bit(1) init('0'b),
      i fixed(15),
      String char(38);

   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);

   put skip list('END ISAM1 TEST');
end CREATE;
```

The input data for this program might look like this:

```
Fred 69 M
Andy 70 M
Susan 72 F
```

Figure 20. Defining and loading a workstation VSAM sequential data set
Using workstation VSAM sequential data sets

**Updating a sequential data set**
The program illustrated in [Figure 20 on page 297](#) can be used to update a workstation VSAM sequential data set. If it is run again, new records are added on the end of the data set.

You can rewrite existing records in a sequential data set, 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 must be sequential record values from a previous WRITE or READ statement.

**Workstation VSAM keyed data sets**
The statements and options allowed for workstation VSAM keyed data sets are shown in Table 18.

<table>
<thead>
<tr>
<th>File declaration</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL OUTPUT BUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); LOCATE based-variable FILE(file-reference) KEYFROM(expression);</td>
<td>SET(pointer-reference)</td>
</tr>
<tr>
<td>SEQUENTIAL OUTPUT UNBUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td>SEQUENTIAL INPUT BUFFERED</td>
<td>READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer reference); READ FILE(file-reference); IGNOR(expression)</td>
<td>KEY(expression) or KEYTO(reference)</td>
</tr>
<tr>
<td>SEQUENTIAL INPUT UNBUFFERED</td>
<td>READ FILE(file-reference) INTO(reference); READ FILE(file-reference); IGNOR(expression)</td>
<td>KEY(expression) or KEYTO(reference)</td>
</tr>
<tr>
<td>SEQUENTIAL UPDATE BUFFERED</td>
<td>READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference); IGNOR(expression)</td>
<td>KEY(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); REWRITE FILE(file-reference); DELETE FILE(file-reference)</td>
<td>FROM(expression) and/or KEY(expression)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KEY(expression)</td>
</tr>
</tbody>
</table>
## Workstation VSAM keyed data sets

### Table 18. Statements and options allowed for loading and accessing workstation VSAM keyed data sets (continued)

<table>
<thead>
<tr>
<th>File declaration¹</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL UPDATE UNBUFFERED</td>
<td>READ FILE(file-reference) INTO(reference); KEY(expression) or KEYTO(reference)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference);¹</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference) FROM(reference); KEY(expression)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE FILE(file-reference); KEY(expression)</td>
<td></td>
</tr>
<tr>
<td>DIRECT³ INPUT BUFFERED</td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td>DIRECT³ INPUT UNBUFFERED</td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td>DIRECT OUTPUT BUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td>DIRECT OUTPUT UNBUFFERED</td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td>DIRECT³ UPDATE BUFFERED</td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference) FROM(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE FILE(file-reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
</tbody>
</table>

¹ Including the key definition statement.
Workstation VSAM keyed data sets

Table 18. Statements and options allowed for loading and accessing workstation VSAM keyed data sets (continued)

<table>
<thead>
<tr>
<th>File declaration¹</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT³ UPDATE UNBUFFERED</td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference) FROM(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE FILE(file-reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

¹ The complete file declaration could 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 DIRECT file with a workstation VSAM data set that has duplicate key capability.

Loading a workstation VSAM keyed data set

When a keyed data set 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.

If a keyed data set already contains some records, and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can add records at the end of the data set only. Again, you must present the records in ascending key order, and you must use the KEYFROM option. In addition, the first record you present must have a key greater than the highest key present on the data set.

Figure 21 on page 301 is an example of a program that loads a workstation VSAM keyed data set. 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 keyed data set must be loaded in this manner. You can also use this program to process a DDM or SFS file by modifying the AMTHD specification.
Workstation VSAM keyed data sets

/* DESCRIPTION */
/* Load an ISAM keyed data set. */
/* */
/* USAGE */
/* The following commands are required to establish */
/* the environment variables to run this program: */
/* */
/* SET DD:DIREC=ISAM2.OUT,AMTHD(ISAM) */
/* SET DD:SYSIN=ISAM2.INP,RECSIZE(80) */
/* */
/* */
NAMELD: proc options(main);
put skip list('START ISAM2 TEST');
dcl Direc file record keyed sequential output
   env(organization(indexed)
      recsize(23)
      keyloc(1)
      keylength(20)
   );
dcl Eof bit(1) init('0'b);
dcl 1 IoArea,
   5 Name char(20),
   5 Number char(3);
on endfile(sysin) Eof = '1'b;
open file(Direc);
get file(sysin) edit(Name,Number) (a(20),a(3));
do while (~Eof);
   write file(Direc) from(IoArea) keyfrom(Name);
   get file(sysin) edit(Name,Number) (a(20),a(3));
end;
close file(Direc);
put skip list('END ISAM2 TEST');
end NAMELD;

Figure 21. Defining and loading a workstation VSAM keyed data set
Workstation VSAM keyed data sets

The input file for this program could be:

<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION, G</td>
<td>162</td>
</tr>
<tr>
<td>BAKER, R</td>
<td>152</td>
</tr>
<tr>
<td>BRAMLEY, O.H.</td>
<td>248</td>
</tr>
<tr>
<td>CHEESMAN, D</td>
<td>141</td>
</tr>
<tr>
<td>CORY, G</td>
<td>336</td>
</tr>
<tr>
<td>ELLIOTT, D</td>
<td>875</td>
</tr>
<tr>
<td>FIGGINS, S</td>
<td>413</td>
</tr>
<tr>
<td>HARVEY, C.O.W.</td>
<td>285</td>
</tr>
<tr>
<td>HASTINGS, G.M.</td>
<td>391</td>
</tr>
<tr>
<td>KENDALL, J.G.</td>
<td>294</td>
</tr>
<tr>
<td>LANCASTER, W.R.</td>
<td>624</td>
</tr>
<tr>
<td>MILES, R</td>
<td>233</td>
</tr>
<tr>
<td>NEWMAN, M.W.</td>
<td>450</td>
</tr>
<tr>
<td>PIT, W.H.</td>
<td>515</td>
</tr>
<tr>
<td>ROLF, D.E.</td>
<td>114</td>
</tr>
<tr>
<td>SHEERS, C.D.</td>
<td>241</td>
</tr>
<tr>
<td>SURCLIFFE, M.</td>
<td>472</td>
</tr>
<tr>
<td>TAYLOR, G.C.</td>
<td>407</td>
</tr>
<tr>
<td>WONG, X-G.</td>
<td>509</td>
</tr>
<tr>
<td>WILTON, L.W.</td>
<td>404</td>
</tr>
<tr>
<td>WINSTONE, E.M.</td>
<td>307</td>
</tr>
<tr>
<td>VISWANATHAN, I.</td>
<td>411</td>
</tr>
</tbody>
</table>

Defining and loading a workstation VSAM keyed data set (continued)

Using a SEQUENTIAL file to access a workstation VSAM keyed data set

You can open a SEQUENTIAL file that is used to access a keyed data set with either the INPUT or the UPDATE attribute.

For READ statements without the KEY option, the records are recovered in ascending key order. 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. This READ statement positions the data set at the specified record; subsequent sequential reads recover the following records in key 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. 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.

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 specified key; otherwise, it is the record that was accessed by the previous READ statement.

Using a DIRECT file to access a workstation VSAM keyed data set

You can open a DIRECT file that is used to access a workstation VSAM keyed data set with the INPUT, OUTPUT, or UPDATE attribute.
Workstation VSAM keyed data sets

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 22 shows one method you can use to update a keyed data set. You can also use this program to process a DDM or SFS file by modifying the AMTHD specification.

```plaintext
/********************************************************************
/* */
/* */
/* DESCRIPTION */
/* Update an ISAM keyed data set by key. */
/* */
/* */
/* USAGE */
/* The following commands are required to establish */
/* the environment variables to run this program: */
/* */
/* SET DD:DIREC=ISAM2.OUT,AMTHD(ISAM) */
/* SET DD:SYSIN=ISAM3.INP,RECSIZE(80) */
/* */
/* Note: This program is using isam2.out file created by the */
/* previous sample program NAMELD. */
/* */
/********************************************************************

DIRUPDT: proc options(main);
  put skip list('ISAM3 TEST START');

dcl Direc file record keyed update
  env(organization(indexed)
     recsize(23)
     keyloc(1)
     keylength(20)
  );

dcl 1 IoArea,
    5 NewArea,
    10 Name char(20),
    10 Number char(3),
    5 Code char(1);

dcl oncode builtin;

dcl Eof bit(1) init('0'b);

on endfile(sysin) Eof = '1'b;

on key(Direc) 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;
```

Figure 22. Updating a workstation VSAM keyed data set
Workstation VSAM keyed data sets

```pli
get file(sysin) edit (Name,Number,Code) (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(NewArea) keyfrom(Name);
    when('C') rewrite file(Direc) from(NewArea) 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) (a(20),a(3),a(1));
end;
close file(Direc);
put file(sysprint) page;

/* Display the updated file */

open file(Direc) sequential input;
Eof = '0'b;
on endfile(Direc) Eof = '1'b;
read file(Direc) into(NewArea);
do while(~Eof);
  put file(sysprint) skip edit(Name,Number)(a,a);
  read file(Direc) into(NewArea);
end;
close file(Direc);

put skip list('ISAM3 TEST END');
end DIRUPDT;

An input file for this program might look like this one:

NEWMAN,M.W. 516C
GOODFELLOW,D.T. 889A
MILES,R. D
HARVEY,C.D.W. 209A
BARTLETT,S.G. 103A
CORY,G. D
READ,K.M. 001A
PITT,W.H. X
ROLF,D.E. D
ELLIOTT,D. 291C
HASTINGS,G.M. D
BRAMLEY,D.H. 439C
GABRIELLI, M. X

Updating a workstation VSAM keyed data set (continued)

The program uses a DIRECT update file and alters the data according to a code
that is passed into the records in the file SYSIN:

A  Add a new record
C  Change the number of an existing name
D  Delete a record

The program reads the name, number, and code and takes action according to the
value of the code. A KEY ON-unit is used to handle any incorrect keys. When the
updating is finished the file DIREC is closed and reopened with the attributes
SEQUENTIAL INPUT. The file is then read sequentially and printed.
Workstation VSAM direct data sets

The statements and options allowed for workstation VSAM direct data sets are:

Table 19. Statements and options allowed for loading and accessing workstation VSAM direct data sets

<table>
<thead>
<tr>
<th>File declaration</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SEQUENTIAL OUTPUT</strong></td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td>KEYFROM(expression) or KEYTO(reference)</td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>LOCATE based-variable FILE(file-reference);</td>
<td>SET(pointer-reference)</td>
</tr>
<tr>
<td><strong>UNBUFFERED</strong></td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td>KEYFROM(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td><strong>SEQUENTIAL INPUT</strong></td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td>KEY(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>READ FILE(file-reference) SET(pointer-reference);</td>
<td>KEY(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td><strong>UNBUFFERED</strong></td>
<td>READ FILE(file-reference); 2</td>
<td>IGNORE(expression)</td>
</tr>
<tr>
<td><strong>SEQUENTIAL UPDATE</strong></td>
<td>READ FILE(file-reference) INTO(reference);</td>
<td>KEY(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>READ FILE(file-reference) SET(pointer-reference);</td>
<td>KEY(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td><strong>UNBUFFERED</strong></td>
<td>READ FILE(file-expression); 2</td>
<td>IGNORE(expression)</td>
</tr>
<tr>
<td><strong>SEQUENTIAL UPDATE</strong></td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td>KEYFROM(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>REWRITE FILE(file-reference);</td>
<td>FROM(reference) and/or KEY(expression)</td>
</tr>
<tr>
<td></td>
<td>DELETE FILE(file-reference);</td>
<td>KEY(expression)</td>
</tr>
<tr>
<td><strong>UNBUFFERED</strong></td>
<td>READ FILE(file-expression); 2</td>
<td>IGNORE(expression)</td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference);</td>
<td>KEYFROM(expression) or KEYTO(expression)</td>
</tr>
<tr>
<td></td>
<td>REWRITE FILE(file-reference);</td>
<td>KEY(expression)</td>
</tr>
<tr>
<td></td>
<td>DELETE FILE(file-reference);</td>
<td>KEY(expression)</td>
</tr>
</tbody>
</table>
### Workstation VSAM direct data sets

**Table 19. Statements and options allowed for loading and accessing workstation VSAM direct data sets (continued)**

<table>
<thead>
<tr>
<th>File declaration</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DIRECT OUTPUT</strong></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECT OUTPUT</strong></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>UNBUFFERED</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECT INPUT</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECT UPDATE</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>BUFFERED</strong></td>
<td>READ FILE(file-reference) INTO(reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ FILE(file-reference) SET(pointer-reference) KEY(expression);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>DIRECT UPDATE</strong></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
<tr>
<td><strong>UNBUFFERED</strong></td>
<td>WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);</td>
<td></td>
</tr>
</tbody>
</table>
Table 19. Statements and options allowed for loading and accessing workstation VSAM direct data sets (continued)

<table>
<thead>
<tr>
<th>File declaration¹</th>
<th>Valid statements, with options you must include</th>
<th>Other options you can also include</th>
</tr>
</thead>
</table>

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.

² The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);

Loading a workstation VSAM direct data set

When a direct data set 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 “Using keys for workstation VSAM data sets” on page 289).

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 a direct data set sequentially, without use of the KEYFROM or KEYTO options, you are not required to use 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.

Figure 23 on page 308 is an example of a program that defines and loads a workstation VSAM direct data set. In the PL/I program, the data set is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used. You can also use this program to process a DDM or SFS file by modifying the AMTHD specification.

If the data were in order and the keys in sequence, it would be possible to use a SEQUENTIAL file and write into the data set from the start. The records would then be placed in the next available slot and given the appropriate number. The number of the key for each record could be returned using the KEYTO option.
Workstation VSAM direct data sets

CREATD: proc options(main);

put skip list('ISAM4 TEST START');

dcl Nos file record output direct keyed
env(organization(relative) recsize(20) );

dcl Sysin file input record;
dcl In_Area,
  2 Name char(20),
  2 Number char(20);
dcl Sysin_Eof bit (1) init('0'b);
dcl Ntemp fixed(15);

on endfile (Sysin) Sysin_Eof = '1'b;

open file(Nos);
read file(Sysin) into(In_Area);
do while(¬Sysin_Eof);
  Ntemp = Number;
  write file(Nos) from(Name) keyfrom(Ntemp);
  put file(sysprint) skip edit (In_Area) (a);
  read file(Sysin) into(In_Area);
end;

close file(Nos);
put skip list('ISAM4 TEST END');

end CREATD;

This could be the input file for this program:

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 23. Loading a workstation VSAM direct data set
Using a SEQUENTIAL file to access a workstation VSAM direct data set

You can open a SEQUENTIAL file that is used to access a direct data set with either the INPUT or the UPDATE attribute. If you use any of the options KEY, KEYTO, or KEYFROM, your file must also use the KEYED attribute.

**Using READ statements**

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 recover the following records in sequence.

**Using WRITE statements**

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.

**Using the REWRITE or DELETE statements**

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 also use DELETE statements, with or without the KEY option, to delete records from the data set.

Using a DIRECT file to access a workstation VSAM direct data set

A DIRECT file used to access a direct data set can have the OUTPUT, INPUT, or UPDATE attribute. You can read, write, rewrite, or delete records exactly as though you were using a KEYED SEQUENTIAL file.

Figure 24 on page 310 shows a direct data set being updated. A DIRECT UPDATE file is used and new records are written by key. You do not need to check for the records being empty, because the empty records are not available under workstation VSAM. You can also use this program to process a DDM or SFS file by modifying the AMTHD specification.

In the second half of the program, the updated file is printed. Again, you do not need to check for the empty records as there is in REGIONAL(1).
Workstation VSAM direct data sets

UPDATD: proc options(main);
   put skip list('ISAM5 TEST START');
   dcl Nos file record keyed
            env(organization(relative));
   dcl Sysin file input record;
   dcl Sysin_Eof bit (1) init('0'b);
   dcl Nos_Eof bit (1) init('0'b);
   dcl 1 In_Area,
       2 Name char(20),
       2 (CNewNo,COldNo) char( 2),
       2 In_Area_1 char( 1),
       2 Code   char( 1);
   dcl IoField char(20);
   dcl NewNo fixed(15);
   dcl OldNo fixed(15);
   dcl oncode builtin;
   on endfile (Sysin) sysin_Eof = '1'b;
   open file (Nos) direct update;

Figure 24. Updating a workstation VSAM direct data set by key

/* DESCRIPTION */
/* Update an ISAM direct data set by key. */
/* */
/* USAGE */
/* The following commands are required to establish */
/* the environment variables to run this program. */
/* */
/* SET DD:SYSIN=ISAM5.INP,RECSIZE(80) */
/* SET DD:NOS=ISAM4.OUT,AMTHD(ISAM),APPEND(Y) */
/* */
/* Note: This sample program is using the direct ISAM data set */
/* isam4.out created by the previous sample program CREATD. */
/* */
/* Figure 24. Updating a workstation VSAM direct data set by key */
/* trap errors */

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;

/* update the direct data set */

read file(Sysin) into(In_Area);
do while(¬Sysin_Eof);
    if CNewNo¬=' ' then
        NewNo = CNewNo;
    else
        NewNo = 0;
    if COldNo¬=' ' then
        OldNo = COldNo;
    else
        OldNo = 0;
    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 list ('Invalid code:',Name);
end;
read file(Sysin) into(In_Area);
end;

close file(Sysin),file(Nos);

/* open and print updated file */

open file(Nos) sequential input;
on endfile (Nos) Nos_Eof = '1'b;

Updating a workstation VSAM direct data set by key (continued)
Workstation VSAM direct data sets

```pli
read file(Nos) into(IoField) keyto(CNewNo);
do while(~Nos_End);
   put file(sysprint) skip
   edit (CNewNo,IoField)(a(5),a);
   read file(Nos) into(IoField) keyto(CNewNo);
   end;
close file(Nos);

put skip list('ISAM5 TEST END');
end UPDATD;
```

An input file for this program might look like this:

```
NEWMAN,M.W. 5640 C
GOODFELLOW,D.T. 89 A
MILES,R. 23 D
HARVEY,C.O.W. 29 A
BARTLETT,S.G. 13 A
CORY,G. 36 D
READ,K.M. 01 A
WONG,X-G. 20 C
PITT,W.H. 55 X
ROLF,D.F. 14 D
ELLIOTT,D. 4285 C
HASTINGS,G.M. 31 D
BRAMLEY,O.H. 4928 C
VISWANATHAN, I. 4115 D
```

*Updating a workstation VSAM direct data set by key (continued)*
Part 5. Using PL/I with databases

Chapter 14. Open Database Connectivity ........................................... 315
  Introducing ODBC ................................................................. 315
    Background ................................................................. 315
    ODBC Driver Manager ..................................................... 315
    Choosing embedded SQL or ODBC ....................................... 315
  Using the ODBC drivers ......................................................... 316
    Online help ...................................................................... 316
    Environment-specific information ....................................... 316
    Driver names .................................................................. 316
    Configuring data sources .................................................. 316
    Connecting to a data source .............................................. 317
    Using a logon dialog box .................................................. 317
    Using a connection string ................................................ 317
    Error messages .................................................................. 317
  ODBC APIs from PL/I ............................................................... 318
    CALL interface convention ................................................ 319
    Using the supplied include files ......................................... 319
    Mapping of ODBC C types .................................................. 320
    Setting licensing information for ODBC Driver Manager/driver .. 321
  Sample program using supplied include files ......................... 321

Chapter 15. Using java Dclgen ......................................................... 323
  Understanding java Dclgen terminology .................................. 323
  PL/I java Dclgen support ....................................................... 324
  Creating a table declaration and host structure ...................... 324
    Selecting a database ........................................................ 325
    Selecting a table and generation a PL/I declaration ............... 325
    Modifying and saving the generated PL/I declaration .......... 326
  Exiting java Dclgen ............................................................. 327
  Including data declarations in your program ......................... 327
Chapter 14. Open Database Connectivity

This chapter contains information to help you use the Open Database Connectivity (ODBC) interface in your PL/I applications. With ODBC, not only can you access data from a variety of databases and file systems that support the ODBC interface, but you can do so dynamically.

Your PL/I applications that use embedded SQL for database access must be processed by a preprocessor for a particular database and have to be recompiled if the target database changes. Because ODBC is a call interface, there is no compile-time designation of the target database as there is with embedded SQL. Not only can you avoid having multiple versions of your application for multiple databases, but your application can dynamically determine which database to target.

Introducing ODBC

ODBC is a specification for an application program interface (API) that enables applications to access multiple database management systems using Structured Query Language (SQL).

ODBC permits maximum interoperability: a single application can access many different database management systems. This enables you to develop, compile, and ship an application without targeting a specific type of data source. Users can then add the database drivers, which link the application to the database management systems of their choice.

Background

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface, referred to as the X/Open Call Level Interface. The goal of this interface is to increase portability of applications by enabling them to become independent of any one database vendor's programming interface.

ODBC was originally developed by Microsoft for Microsoft operating systems based on a preliminary draft of X/Open CLI. Since this time, other vendors have provided ODBC drivers that run on other platforms, such as OS/2 and UNIX systems.

The descriptions and examples in this chapter apply to ODBC Version 3.0. For detailed information about ODBC include files, see “Using the supplied include files” on page 319.

ODBC Driver Manager

When you use the ODBC interface, your application makes calls through a Driver Manager. The Driver Manager dynamically loads the necessary driver for the database server to which the application connects. The driver, in turn, accepts the call, sends the SQL to the specified data source (database), and returns any result.

Choosing embedded SQL or ODBC

Embedded SQL and ODBC have advantages particular to them. Some of the advantages of embedded SQL are:
Introducing ODBC

- Static SQL usually provides better performance than dynamic SQL. It does not have to be prepared at run time, thus reducing both processing and network traffic.
- With static SQL, database administrators have to grant users access to a package only rather than access to each table or view that is used.

Some of the advantages of ODBC are:
- It provides a consistent interface regardless of what kind of database server is used.
- You can have more than one concurrent connection.
- Applications do not have to be bound to each database on which they execute. Although PL/I for Windows does this bind for you automatically, it binds automatically to only one database. If you want to choose which database to connect to dynamically at run time, you must take extra steps to bind to a different database.

Using the ODBC drivers

To enable ODBC for data access in PL/I, you must install the ODBC Driver Manager and drivers by selecting the “ODBC Drivers” component during installation.

**Important:** During the installation process, a license file for the ODBC driver is installed on your system.

A file named ivib.lic is installed in \plidir\ODBC, where \ and plidir are the drive and directory respectively, where PL/I for Windows is installed.

You must keep this file in the install directory because it is used when you run your application to verify that you are licensed to use the ODBC driver. In “Setting licensing information for ODBC Driver Manager/driver” on page 321 you learn how to use a function call to trigger the verification.

Online help

Online help is available for the ODBC drivers, both as a reference book and as context-sensitive help. The specific file names and so on may differ; you should note the names given in this section for the file names for PL/I.

Environment-specific information

The ODBC drivers are 32-bit drivers. The required network software supplied by your database system vendors must be 32-bit compliant.

Driver names

The drivers for Windows should be at the ODBC 3.0 level or higher. ODBC.INI is a subkey of the HKEY_CURRENT_USER\SOFTWARE\ODBC key in the Windows registry. The ODBC.INI subkey is maintained by the ODBC Administrator, which is located in the main PL/I program group. Since Windows can support multiple users, the ODBC.INI subkey is stored under unique user keys in the registry.

Configuring data sources

A data source consists of a DBMS and any remote operating system and network necessary to access it. After the drivers have been installed, the data source must be configured using the ODBC Administrator program, which is located in the
main PL/I program group. Because Windows can host multiple users, each user
must configure their own data sources. For detailed configuration information for
the specific driver you wish to configure, refer to the appropriate section of the
on-line help.

Connecting to a data source

Your ODBC application needs to connect to the data source either using a logon
dialog box or a connection string, depending on the data source.

Using a logon dialog box

Some ODBC applications display a logon dialog box when you are connecting to a
data source. In these cases, the data source name has already been specified.

In the logon dialog box, do the following:

1. Type the name of the remote database or select the name of the remote
database from the Database Name drop-down list.
   You must have cataloged any database you want to access from the client.
2. If required, type your user name (authorization ID).
3. If required, type your password.
   If you leave your user name and password blank, the ODBC application
   assumes you have already logged on using SQLLOGN2 (under DOS) or using
   User Profile Management. If you have not, the application returns an error. You
   must either type your user name and password in the dialog box or log on
   using SQLLOGN2 and STARTDRQ (under DOS) or using User Profile
   Management.
4. Click OK to complete the logon and to update the values in ODBC.INI.

Using a connection string

If your application requires a connection string to connect to a data source, you
must specify the data source name that tells the driver which ODBC.INI section to
use for the default connection information. Optionally, you may specify
attribute=value pairs in the connection string to override the default values
stored in ODBC.INI. These values are not written to ODBC.INI.

You can specify either long or short names in the connection string. The connection
string has the form:

DSN=data_source_name[;attribute=value[;attribute=value]...]

An example of a connection string for INFORMIX 5 is

DSN=INFORMIX TABLES;DB=PAYROLL

Error messages

Error messages can come from the following sources:
• An ODBC driver
• The database system
• The Driver Manager.

An error reported on an ODBC driver has the following format:

[vendor] [ODBC_component] message

ODBC_component is the component in which the error occurred. For example, an
error message from INTERSOLV's SQL Server driver would look like this:

[INTERSOLV] [ODBC SQL Server driver] Login incorrect.
Introducing ODBC

If you get this type of error, check the last ODBC call your application made for possible problems or contact your ODBC application vendor.

An error that occurs in the data source includes the data source name, in the following format:
[vendor] [ODBC_component] [data_source] message

With this type of message, ODBC_component is the component that received the error from the data source indicated. For example, you may get the following message from an Oracle data source:
[INTER SOLV] [ODBC Oracle driver] [Oracle] ORA-0919: specified length too long for CHAR column

If you get this type of error, you did something incorrectly with the database system. Check your database system documentation for more information or consult your database administrator. In this example, you would check your Oracle documentation.

The Driver Manager is an application that establishes connections with drivers, submits requests to drivers, and returns results to applications. An error that occurs in the Driver Manager has the following format:
[vendor] [ODBC DLL] message

vendor can be Microsoft or INTER SOLV. For example, an error from the Microsoft Driver Manager might look like this:
[Microsoft] [ODBC DLL] Driver does not support this function

ODBC APIs from PL/I

Included with VA PL/I are ODBC include files that make it easier for you to access data bases with ODBC drivers using ODBC calls from your PL/I programs. This section describes the supplied ODBC include files, how ODBC API argument types map to PL/I data descriptions, and additional PL/I functions and considerations applicable to ODBC APIs.

For details on the ODBC APIs, see the online help.

For specific information related to an ODBC driver, such as the ODBC level or extensions supported by that driver, please refer to the specifications available with that driver.

The following illustrate how to access ODBC from PL/I programs:
- "CALL interface convention" on page 319
- "Using the supplied include files" on page 319
- "Mapping of ODBC C types" on page 320
- "Setting licensing information for ODBC Driver Manager/driver" on page 321

LIB Files:

When you link your ODBC applications, you must include the import library ODBC32.LIB, which is included in the ODBC SDK (from Microsoft).
CALL interface convention

Programs making ODBC calls must be compiled with the DEFAULT(BYVALUE) and LIMITS(EXTNAME(31)) compile-time options.

Using the supplied include files

The include files described and listed here are for ODBC Version 3.0.

Table 20. Supplied include files for ODBC

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODBCSQL.CPY</td>
<td>Main include for ODBC functions</td>
</tr>
<tr>
<td>ODBCEXT.CPY</td>
<td>Include for Microsoft's ODBC extensions</td>
</tr>
<tr>
<td>ODBCTYPE.CPY</td>
<td>Include for ODBC type definitions</td>
</tr>
<tr>
<td>ODBCUCOD.CPY</td>
<td>Include unicode</td>
</tr>
<tr>
<td>ODBCSAMP.PLI</td>
<td>Sample program</td>
</tr>
</tbody>
</table>

The supplied include files define the symbols for constant values described for ODBC APIs, mapping constants used in calls to ODBC APIs to symbols specified in ODBC guides so that argument (input and output) and function return values can be specified and tested. These files should be included in your PL/I program in order to use ODBC API calls.

In PL/I, names longer than 31 characters are truncated or abbreviated to 31 characters. Table 21 shows the names that are longer than 31 characters, and their corresponding PL/I names.

Table 21. ODBC names truncated or abbreviated for PL/I

<table>
<thead>
<tr>
<th>ODBC C #define symbol &gt; 31 characters long</th>
<th>Corresponding PL/I name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_AD_ADD_CONSTRAINT_DEFERRABLE</td>
<td>SQL_AD_ADD_CONSTR_DEFERRABLE</td>
</tr>
<tr>
<td>SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED</td>
<td>SQL_AD_ADD_CONSTR_INITLY_DEFERD</td>
</tr>
<tr>
<td>SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE</td>
<td>SQL_AD_ADD_CONSTR_INITLY_IMMED</td>
</tr>
<tr>
<td>SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE</td>
<td>SQL_AD_ADD_CONSTR_NON_DEFERRABL</td>
</tr>
<tr>
<td>SQL_API_ODBC3_ALL_FUNCTIONS_SIZE</td>
<td>SQL_API_ODBC3_ALL_FUNCTIONS_SZ</td>
</tr>
<tr>
<td>SQL_AT_CONSTRAINT_INITIALLY_DEFERRED</td>
<td>SQL_AT_CONSTR_INITIALLY_DEFERD</td>
</tr>
<tr>
<td>SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE</td>
<td>SQL_AT_CONSTR_INITIALLY_IMMED</td>
</tr>
<tr>
<td>SQL_AT_CONSTRAINT_NAME_DEFINITION</td>
<td>SQL_AT_CONSTR_NAME_DEFINITION</td>
</tr>
<tr>
<td>SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE</td>
<td>SQL_AT_DROP_TBL_CONSTR_CASCADE</td>
</tr>
<tr>
<td>SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT</td>
<td>SQL_AT_DROP_TBL_CONSTR_RESTRICT</td>
</tr>
<tr>
<td>SQL_CA_CONSTRAINT_INITIALLY_DEFERRED</td>
<td>SQL_CA_CONSTR_INITLY_DEFERED</td>
</tr>
<tr>
<td>SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE</td>
<td>SQL_CA_CONSTR_INITLY_IMMEDIATE</td>
</tr>
<tr>
<td>SQL_CA_CONSTRAINT_NON_DEFERRABLE</td>
<td>SQL_CA_CONSTR_NON_DEFERRABLE</td>
</tr>
<tr>
<td>SQL_CDO_CONSTRAINT_NAME_DEFINITION</td>
<td>SQL_CDO_CONSTR_NAME_DEFINITION</td>
</tr>
<tr>
<td>SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED</td>
<td>SQL_CDO_CONSTR_INITLY_DEFERED</td>
</tr>
<tr>
<td>SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE</td>
<td>SQL_CDO_CONSTR_INITLY_IMMEDAT</td>
</tr>
<tr>
<td>SQL_CDO_CONSTRAINT_NON_DEFERRABLE</td>
<td>SQL_CDO_CONSTR_NON_DEFERRABLE</td>
</tr>
</tbody>
</table>
### Table 21. ODBC names truncated or abbreviated for PL/I (continued)

<table>
<thead>
<tr>
<th>ODBC C #define symbol &gt; 31 characters long</th>
<th>Corresponding PL/I name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_CT_CONSTRAINT_INITIALLY_DEFERRED</td>
<td>SQL_CT_CONSTR_INITLY_DEFERRED</td>
</tr>
<tr>
<td>SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE</td>
<td>SQL_CT_CONSTR_INITLY_IMMEDIATE</td>
</tr>
<tr>
<td>SQL_CT-unstyled</td>
<td>SQL_CT_CONSTR Non-Deferrable</td>
</tr>
<tr>
<td>SQL_CT-unstyled</td>
<td>SQL_CT_CONSTR_NAME_DEFINITION</td>
</tr>
<tr>
<td>SQL_DESC_DATE</td>
<td>SQL_DESC_DATE_TIME_INTERVAL_PRECISION</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR</td>
<td>SQL_DL_SQL92_INTERVAL_DAY_TO_HR</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE</td>
<td>SQL_DL_SQL92_INTERVAL_DAY_TO_MIN</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND</td>
<td>SQL_DL_SQL92_INTERVAL_DAY_TO_SEC</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE</td>
<td>SQL_DL_SQL92_INTERVAL_HOUR_TO_MIN</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND</td>
<td>SQL_DL_SQL92_INTERVAL_HOUR_TO_SEC</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND</td>
<td>SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND</td>
</tr>
<tr>
<td>SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH</td>
<td>SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1</td>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIB1</td>
</tr>
<tr>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2</td>
<td>SQL_FORWARD_ONLY_CURSOR_ATTRIB2</td>
</tr>
<tr>
<td>SQL_MAX_ASYNC_CONCURRENT_STATEMENTS</td>
<td>SQL_MAX_ASYNC_CONCURRENT_STMTS</td>
</tr>
<tr>
<td>SQL_MAXIMUM_CONCURRENT_ACTIVITIES</td>
<td>SQL_MAXIMUM_CONCURRENT_ACTIVITIES</td>
</tr>
<tr>
<td>SQL_SQL92_FOREIGN_KEY_DELETE_RULE</td>
<td>SQL_SQL92_FOREIGN_KEY_DELETE_RULE</td>
</tr>
<tr>
<td>SQL_SQL92_FOREIGN_KEY_UPDATE_RULE</td>
<td>SQL_SQL92_FOREIGN_KEY_UPDATE_RULE</td>
</tr>
<tr>
<td>SQL_SQL92_NUMERIC_VALUE_FUNCTIONS</td>
<td>SQL_SQL92_NUMERIC_VALUE_FUNCTIONS</td>
</tr>
<tr>
<td>SQL_SQL92_RELATIONAL_JOIN_OPERATORS</td>
<td>SQL_SQL92_RELATIONAL_JOIN_OPER</td>
</tr>
<tr>
<td>SQL_TRANSACTION_ISOLATION_OPTION</td>
<td>SQL_TRANSACTION_ISOLATION_OPTN</td>
</tr>
<tr>
<td>SQL_TRANSACTION_READ_UNCOMMITTED</td>
<td>SQL_TRANSACTION_READ_UNCOMMITTD</td>
</tr>
</tbody>
</table>

### Mapping of ODBC C types

The data types specified in ODBC APIs are defined in terms of ODBC C types in the API definitions. The following table shows corresponding PL/I declarations for the indicated ODBC C types of the arguments.

<table>
<thead>
<tr>
<th>ODBC C type</th>
<th>PL/I form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLSMALLINT</td>
<td>FIXED BIN(15)</td>
<td>Signed short integer (2 byte binary)</td>
</tr>
<tr>
<td>SQLUSMALLINT</td>
<td>FIXED BIN(16) / UNSIGNED</td>
<td>Unsigned short integer (2 byte binary)</td>
</tr>
<tr>
<td>SQLINTEGER</td>
<td>FIXED BIN(31) / UNSIGNED</td>
<td>Signed long integer (4 byte binary)</td>
</tr>
<tr>
<td>SQLUNINTEGER</td>
<td>FIXED BIN(31) / UNSIGNED</td>
<td>Unsigned long integer (4 byte binary)</td>
</tr>
<tr>
<td>SQLREAL</td>
<td>FLOAT</td>
<td>Floating point (4 bytes)</td>
</tr>
<tr>
<td>SQLFLOAT</td>
<td>DOUBLE</td>
<td>Floating point (8 bytes)</td>
</tr>
<tr>
<td>SQLDOUBLE</td>
<td>DOUBLE</td>
<td>Floating point (8 bytes)</td>
</tr>
<tr>
<td>SQLCHAR *</td>
<td>CHAR(*) VARZ BYADDR</td>
<td>Pointer to unsigned character.</td>
</tr>
<tr>
<td>SQLHDBC</td>
<td>POINTER</td>
<td>Connection handle</td>
</tr>
</tbody>
</table>
**ODBC APIs from PL/I**

### Table 22. Mapping of ODBC C Type to PL/I Data Declarations (continued)

<table>
<thead>
<tr>
<th>ODBC C type</th>
<th>PL/I form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLHENV</td>
<td>POINTER</td>
<td>Environment handle</td>
</tr>
<tr>
<td>SQLHSTMT</td>
<td>POINTER</td>
<td>Statement handle</td>
</tr>
<tr>
<td>SQLHWND</td>
<td>POINTER</td>
<td>Window handle</td>
</tr>
</tbody>
</table>

### Setting licensing information for ODBC Driver Manager/driver

When using the ODBC Driver Manager/drivers, you need to call `ibmODBCLicInfo` immediately following a call to the `SQLConnect`, `SQLDriverConnect`, or `SQLBrowseConnect` functions. You need to pass the argument 'hdbc' to `ibmODBCLicInfo` like this:

```plaintext```
sql_rc = ibmODBCLicInfo(myHDBC);
```

The `ibmODBCLicInfo` routine is included in the `ibmodlic.lib` library which must be included in the link step of your program. Refer to the sample program, `odbc samp.pli` for more information.

### Sample program using supplied include files

A sample PL/I program is supplied illustrating the use of some common ODBC functions, including:

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocEnv</td>
</tr>
<tr>
<td>SQLAllocConnect</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
</tr>
<tr>
<td>SQLBindCol</td>
</tr>
<tr>
<td>SQLBindParameter</td>
</tr>
<tr>
<td>SQLConnect</td>
</tr>
<tr>
<td>SQLDisconnect</td>
</tr>
<tr>
<td>SQLError</td>
</tr>
<tr>
<td>SQLExecDirect</td>
</tr>
<tr>
<td>SQLExecute</td>
</tr>
<tr>
<td>SQLFetch</td>
</tr>
<tr>
<td>SQLFreeConnect</td>
</tr>
<tr>
<td>SQLFreeEnv</td>
</tr>
<tr>
<td>SQLFreeStmt</td>
</tr>
<tr>
<td>SQLGetInfo</td>
</tr>
<tr>
<td>SQLNativeSQL</td>
</tr>
<tr>
<td>SQLPrepare</td>
</tr>
<tr>
<td>SQLTransact</td>
</tr>
</tbody>
</table>

### Example Notes:

1. Use the `DEFAULT(BYVALUE)` and `LIMITS(EXTNAME(31))` options to compile ODBC programs.
2. For Windows, a sample PL/I program is supplied in the `..\samples\` directory. Use the command file `bldodbc.bat` found in the same directory to compile and link the test program.
3. The ODBC include files are available in the `\include\` subdirectory.
Sample program
Chapter 15. Using java Dclgen

PL/I for Windows comes with a declarations generator (java Dclgen) that produces DECLARE statements you can use in your PL/I applications.

java Dclgen users

In order to use java Dclgen in the Windows environment, you must have the Java Developer's Toolkit (V1.3 or later) and DB2 installed on your system.

The java Dclgen tool:
- Generates a table declaration and puts it into a file that you can include in your program.
- Gets information about the definition of the table and each column within the table from the database catalog.
- The java Dclgen now supports Lightweight Directory Access Protocol (LDAP) directory services when you upgrade to DB2 Universal Database Version 8 or later.
- Uses the information to produce a complete SQL DECLARE statement for the table (or view) and a matching PL/I structure declaration.

To use the declarations in your program, use the SQL INCLUDE statement.

If you wish to invoke java Dclgen and your table names include DBCS characters, you need to use a terminal that can input and display double-byte characters.

Understanding java Dclgen terminology

The following information explains the terms used in java Dclgen dialog boxes:

Tables
- The unqualified table name for which you want java Dclgen to produce SQL data declarations. Optionally, you can qualify the table name by entering the table qualifier in the Table Qualifier entry field. The tool generates a two-part table name from the table name and table qualifier.

Table qualifier
- The table name qualifier. If you do not specify this value, your logon ID is assumed to be the table qualifier.

Output Path for Save
- The path targeted for the declarations that java Dclgen produces.

Output Filename for Save
- The filename targeted for the declarations that java Dclgen produces.

Structure name
- Name of the generated data structure which can be up to 31 characters in length.
- If you leave this field blank, java Dclgen generates a name that contains the table or view name with a DCL prefix. If the table or view name consists of a DBCS string, the prefix consists of DBCS characters.
Understanding java Dclgen terminology

Field Name Prefix
Prefix name generated for fields in the javaDclgen output. The value you choose can be up to 28 characters in length and is used as the prefix for the field name.

For example, if you choose ABCDE, the field names generated are ABCDE001, ABCDE002, and so on.

If you leave this field blank, the field names are the same as the column names in the table or view. If the name is a DBCS string, DBCS equivalents of the suffix numbers are generated.

A table or column name in the DECLARE statement is generated as a non-delimited identifier unless the name contains special characters and is not a DBCS string.

If you are using an SQL reserved word as an identifier, you must edit the java Dclgen output in order to add the appropriate SQL delimiters.

PL/I java Dclgen support

Variable names and data attributes generated by java Dclgen are derived from the information contained in databases.

Table 23. Declarations generated by java Dclgen

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>PL/I</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>BIN FIXED(15)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>BIN FIXED(31)</td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>DEC FIXED(p,s)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>BIN FLOAT(53)</td>
</tr>
<tr>
<td>CHAR(1)</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>CHAR(n) VARYING</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>CHAR(32700) VARYING</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>GRAPHIC(n)</td>
</tr>
<tr>
<td>VARCHARGRAPHIC(n)</td>
<td>GRAPHIC(n) VARYING</td>
</tr>
<tr>
<td>LONG VARCHARGRAPHIC</td>
<td>GRAPHIC(16350) VARYING</td>
</tr>
<tr>
<td>DATE</td>
<td>CHAR(10)</td>
</tr>
<tr>
<td>TIME</td>
<td>CHAR(8)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>CHAR(26)</td>
</tr>
<tr>
<td>CLOB(nnn)</td>
<td>SQL TYPE IS CLOB(nnn)</td>
</tr>
<tr>
<td>BLOB(nnn)</td>
<td>SQL TYPE IS BLOB(nnn)</td>
</tr>
<tr>
<td>DBCLOB(nnn)</td>
<td>SQL TYPE IS DBCLOB(nnn)</td>
</tr>
</tbody>
</table>

Creating a table declaration and host structure

You can start java Dclgen in one of two ways:
1. Enter java javaDclgen at the MS/DOS prompt.
2. Double-click on the java Dclgen icon in the main PL/I program group.
Selecting a database

A window appears and gives you a list of available databases in the Databases list box. To select a database, move your mouse pointer to the database entry and click your left mouse button once. This should highlight your selection.

Just below the Databases list box is the Table Qualifier (Required) entry field.

![Image of Java Dclgen window]

Figure 25. Selecting a database

This field is filled in with the current user's ID. You can use this default table qualifier or you can replace it with another valid table qualifier.

To continue, click on the Get a List of Tables for the Database button.

Selecting a table and generation a PL/I declaration

The Tables list box should be populated with the tables created in the database by the table qualifier. You can select a table in the database by clicking on it with your mouse pointer.
Creating a table declaration and host structure

You can also choose to specify a level 1 name in the **Structure Name** field as well as a field name prefix to be used in each level 2 name in the structure. For example, if you specify MYSTRUCT as the field name prefix, the level 2 names are MYSTRUCT001, MYSTRUCT002, and so on.

Click on the **Generate Include Structure for Table** button to continue.

**Modifying and saving the generated PL/I declaration**

The next window you should see has a text area containing the generated PL/I declaration. You can edit the contents of this area directly if needed.
Creating a table declaration and host structure

For PL/I, java Dclgen saves the contents of the text area in a file in the ibmpliw\include using the table name as the filename with an extension of .CPY.

If you decide to save the generated declaration somewhere other than this directory, click on the Change Location and Save button. You can change the output directory and filename using the Save As... dialog.

You can change the table qualifier or editor name (including any extensions) by typing over the default information.

Note: If the table name contains any special characters that are not part of a filename, you should specify a new filename.

Exiting java Dclgen

To exit or quit java Dclgen, click on the End buttons successively until the application ends.

Including data declarations in your program

Use the following SQL INCLUDE statement to insert the table declaration and PL/I structure declaration produced by java Dclgen into your source program:

```
exec sql
   include name ;
```

For example, to include a description for the table BALKBJ.ORG, code:

```
exec sql
   include org ;
```

If for some reason java Dclgen produces some unexpected results, you can use the editor to tailor the output to your specific needs.
Creating a table declaration and host structure
## Part 6. Advanced topics

### Chapter 16. Using the Program Maintenance Utility, NMAKE

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why use NMAKE?</td>
<td>333</td>
</tr>
<tr>
<td>Running NMAKE</td>
<td>333</td>
</tr>
<tr>
<td>Using the command line</td>
<td>333</td>
</tr>
<tr>
<td>Command-line syntax</td>
<td>333</td>
</tr>
<tr>
<td>Command-line help</td>
<td>334</td>
</tr>
<tr>
<td>Using NMAKE command files</td>
<td>334</td>
</tr>
<tr>
<td>Why use a command file?</td>
<td>334</td>
</tr>
<tr>
<td>Command file syntax</td>
<td>335</td>
</tr>
<tr>
<td>Example</td>
<td>335</td>
</tr>
<tr>
<td>NMAKE options</td>
<td>335</td>
</tr>
<tr>
<td>Produce error file (/X)</td>
<td>335</td>
</tr>
<tr>
<td>Build all targets (/A)</td>
<td>335</td>
</tr>
<tr>
<td>Suppress messages (/C)</td>
<td>335</td>
</tr>
<tr>
<td>Display modification dates (/D)</td>
<td>336</td>
</tr>
<tr>
<td>Override environment variables (/E)</td>
<td>336</td>
</tr>
<tr>
<td>Specify description file (/F)</td>
<td>336</td>
</tr>
<tr>
<td>Display help (/HELP or /?)</td>
<td>336</td>
</tr>
<tr>
<td>Ignore exit codes (/I)</td>
<td>336</td>
</tr>
<tr>
<td>Display commands (/N)</td>
<td>336</td>
</tr>
<tr>
<td>Suppress sign-on banner (/NOLOGO)</td>
<td>336</td>
</tr>
<tr>
<td>Print macro and target definitions (/P)</td>
<td>337</td>
</tr>
<tr>
<td>Return exit code (/Q)</td>
<td>337</td>
</tr>
<tr>
<td>Ignore TOOLS.INI file (/R)</td>
<td>337</td>
</tr>
<tr>
<td>Suppress command display (/S)</td>
<td>337</td>
</tr>
<tr>
<td>Change target modification dates (/T)</td>
<td>337</td>
</tr>
<tr>
<td>Description files</td>
<td>337</td>
</tr>
<tr>
<td>Description blocks</td>
<td>338</td>
</tr>
<tr>
<td>Special features</td>
<td>338</td>
</tr>
<tr>
<td>Targets in several description blocks</td>
<td>339</td>
</tr>
<tr>
<td>Using macros</td>
<td>340</td>
</tr>
<tr>
<td>Macros example</td>
<td>340</td>
</tr>
<tr>
<td>Special features</td>
<td>340</td>
</tr>
<tr>
<td>Macros in a description file</td>
<td>340</td>
</tr>
<tr>
<td>Macros on the command line</td>
<td>341</td>
</tr>
<tr>
<td>Inherited macros</td>
<td>341</td>
</tr>
<tr>
<td>Defined macros</td>
<td>341</td>
</tr>
<tr>
<td>Macro substitutions</td>
<td>342</td>
</tr>
<tr>
<td>Special macros</td>
<td>342</td>
</tr>
<tr>
<td>Special macros examples</td>
<td>343</td>
</tr>
<tr>
<td>File-specification parts</td>
<td>344</td>
</tr>
<tr>
<td>Characters that modify special macros</td>
<td>344</td>
</tr>
<tr>
<td>Modified special macros example</td>
<td>344</td>
</tr>
<tr>
<td>Macro precedence rules</td>
<td>344</td>
</tr>
<tr>
<td>Inference rules</td>
<td>345</td>
</tr>
<tr>
<td>Special features</td>
<td>346</td>
</tr>
<tr>
<td>Inference rules example</td>
<td>346</td>
</tr>
<tr>
<td>Inference-rule path specifications</td>
<td>346</td>
</tr>
<tr>
<td>Predefined inference rules</td>
<td>347</td>
</tr>
<tr>
<td>Directives</td>
<td>347</td>
</tr>
<tr>
<td>Directives example</td>
<td>349</td>
</tr>
<tr>
<td>Pseudotargets</td>
<td>349</td>
</tr>
<tr>
<td>Predefined pseudotargets</td>
<td>349</td>
</tr>
<tr>
<td>.SILENT Pseudotarget</td>
<td>349</td>
</tr>
<tr>
<td>.IGNORE Pseudotarget</td>
<td>349</td>
</tr>
<tr>
<td>.SUFFIXES Pseudotarget</td>
<td>350</td>
</tr>
<tr>
<td>.PRECIOUS Pseudotarget</td>
<td>350</td>
</tr>
<tr>
<td>Inline files</td>
<td>350</td>
</tr>
<tr>
<td>Inline files example</td>
<td>351</td>
</tr>
<tr>
<td>Escape characters</td>
<td>351</td>
</tr>
<tr>
<td>Characters that modify commands</td>
<td>352</td>
</tr>
<tr>
<td>Turn error checking off (-)</td>
<td>352</td>
</tr>
<tr>
<td>Dash command modifier examples</td>
<td>352</td>
</tr>
<tr>
<td>Suppress command display (@)</td>
<td>353</td>
</tr>
<tr>
<td>At sign (@) command modifier example</td>
<td>353</td>
</tr>
<tr>
<td>Execute command for dependents (!)</td>
<td>353</td>
</tr>
<tr>
<td>Exclamation point (!) command modifier examples</td>
<td>353</td>
</tr>
<tr>
<td>EXTMAKE Syntax</td>
<td>354</td>
</tr>
<tr>
<td>Macros and inference rules in TOOLS.INI</td>
<td>354</td>
</tr>
<tr>
<td>TOOLS.INI example</td>
<td>354</td>
</tr>
</tbody>
</table>

### Chapter 17. Improving performance

Selecting compile-time options for optimal performance | 355 |
| OPTIMIZE | 355 |
| IMPRECISE | 355 |
| GONUMBER | 356 |
| SNAP | 356 |
| RULES | 356 |
| PREFIX | 357 |
| CONVERSION | 357 |
| FIXEDOVERFLOW | 357 |
| DEFAULT | 357 |
| BYADDR or BYVALUE | 358 |
| (NON)CONNECTED | 359 |
| RETURNS(BYVALUE) or RETURNS(BYADDR) | 359 |
| (NO)DESCRIPTOR | 359 |
| (RE)ORDER | 359 |
| LINKAGE | 359 |
| ASCII or EBCDIC | 360 |
| IEEE or HEXADEC | 360 |
| (NON)NATIVE | 360 |
| (NO)INLINE | 360 |
| Summary of compile-time options that improve performance | 361 |
| Coding for better performance | 361 |
| DATA-directed input and output | 361 |
| Input-only parameters | 362 |
| String assignments | 362 |
| Loop control variables | 362 |
| PACKAGES versus nested PROCEDUREs | 363 |
| Example with nested procedures | 363 |
| REDUCIBLE functions | 364 |
| DEFINED versus UNION | 364 |
| Named constants versus static variables | 365 |
| Example with optimal code but no meaningful names | 365 |
| Avoiding calls to library routines | 366 |
Chapter 18. Using user exits
Using the compiler user exit
  Procedures performed by the compiler user exit
  Activating the compiler user exit
  Customizing the compiler user exit
    Modifying IBMUEXIT.INF
    Writing your own compiler exit
    Structure of global control blocks
    Writing the initialization procedure
    Writing the message filtering procedure
    Writing the termination procedure
Using the CICS run-time user exit
  Prior to program invocation
  After program termination
  Modifying CEEFXITA
Using data conversion tables

Chapter 19. Building dynamic link libraries
Creating DLL source files
Compiling your DLL source
Preparing to link your DLL
  Specifying exported names under Windows
Linking your DLL
  Using your DLL
Sample program to build a DLL
Using FETCH and RELEASE in your main program
Exporting data from a DLL

Chapter 20. Using IBM Library Manager on Windows
Running ILIB
  Using the command line
  Using the ILIB environment variable
    Command line
    Windows control panel
    Windows 98 AUTOEXEC.BAT file
  Using an ILIB response file
Examples specifying ILIB parameters
Controlling ILIB input
Controlling ILIB output
Controlling ILIB output
ILIB objects
  Summary of ILIB objects
  Add/Replace
  /EXTRACT
  /REPLACE
ILIB options
  Summary of ILIB options
  /?
  /BACKUP
  /DEF
  /FREEFORMAT
  /GENDEF
  /GI
  /HELP
  /LIST
  /NOEXT
  /OUT
  /QUIET
/WARN

Chapter 21. Calling conventions
Understanding linkage considerations
OPTLINK linkage
  Features of OPTLINK
  Tips for using OPTLINK
General-purpose register implications
Parameters
Examples of passing parameters
  Passing conforming parameters to a routine
SYSTEM linkage
  Features of SYSTEM
  Example using SYSTEM linkage
STDCALL linkage (Windows only)
  Features of STDCALL
Examples using the STDCALL convention
Using WinMain (Windows only)
CDECL linkage
  Features of CDECL
Examples using the CDECL convention

Chapter 22. Using PL/I in mixed-language applications
Matching data and linkages
  What data is passed
  How data is passed
  Where data is passed
Maintaining your environment
Invoking non-PL/I routines from a PL/I MAIN
Invoking PL/I routines from a non-PL/I main
Using ON ANYCONDITION

Chapter 23. Interfacing with Java
What is the Java Native Interface (JNI)?
JNI Sample Program #1 - 'Hello World'
  Step 1: Writing the Java Program
    Declare the Native Method
    Load the Native Library
    Write the Java Main Method
  Step 2: Compiling the Java Program
  Step 3: Writing the PL/I Program
    Useful PL/I Compiler Options
Correct Form of PL/I Procedure Name and Procedure Statement
JNI Include File
The Complete PL/I Procedure
  Step 4: Compiling and Linking the PL/I Program
Compiling the PL/I Program
Linking the Dynamic Link Library
  Step 5: Running the Sample Program
JNI Sample Program #2 - Passing a String
  Step 1: Writing the Java Program
    Declare the Native Method
    Load the Native Library
    Write the Java Main Method
  Step 2: Compiling the Java Program
  Step 3: Writing the PL/I Program
Correct Form of PL/I Procedure Name and Procedure Statement

PL/I for Windows Programming Guide
JNI Include File
The Complete PL/I Procedure.
Step 4: Compiling and Linking the PL/I Program
Compiling the PL/I Program
Linking the Dynamic Link Library
Step 5: Running the Sample Program
JNI Sample Program #3 - Passing an Integer
Step 1: Writing the Java Program
Declare the Native Method
Load the Native Library
Write the Java Main Method
Step 2: Compiling the Java Program
Step 3: Writing the PL/I Program
Correct Form of PL/I Procedure Name and Procedure Statement
JNI Include File
The Complete PL/I Procedure.
Step 4: Compiling and Linking the PL/I Program
Compiling the PL/I Program
Linking the Dynamic Link Library
Step 5: Running the Sample Program
Determining equivalent Java and PL/I data types

Chapter 24. Using sort routines
Comparing S/390 and workstation sort programs
Preparing to use sort
Choosing the type of sort
Specifying the sorting field
Example:
Specifying the records to be sorted
Example:
Calling the sort program
PLISRT examples
Example 1
Example 2
Example 3
Example 4
Determining whether the sort was successful
Getting diagnostic information
The IBM_SORT_RC environment variable
The IBM_SORT_DIR environment variable
The IBM_SORT_MSGFILE environment variable
Sort data input and output
Sort data handling routines
E15 — input-handling routine (sort exit E15)
E35 — output-handling routine (sort exit E35)
Calling PLISRTA
Calling PLISRTB
Calling PLISRTC
Calling PLISRTD, example 1
Calling PLISRTD, example 2

Chapter 25. Using the SAX parser
Overview
The PLISAXA built-in subroutine
The PLISAXB built-in subroutine
The SAX event structure
start_of_document
version_information
encoding_declaration
standalone_declaration
document_type_declaration
end_of_document
start_of_element
attribute_name
attribute_characters
attribute_predefined_reference
attribute_character_reference
end_of_element
start_of_CDATA_section
end_of_CDATA_section
content_characters
content_predefined_reference
content_character_reference
processing_instruction
comment
unknown_attribute_reference
unknown_content_reference
start_of_prefix_mapping
end_of_prefix_mapping
exception
Parameters to the event functions
Coded character sets for XML documents
Supported EBCDIC code pages
Supported ASCII code pages
Specifying the code page
Using a number:
Using an alias:
Exceptions
Example
Continuable exception codes
Terminating exception codes

Chapter 26. Using PL/I MLE in your applications
Applying attributes and options
DATE attribute
RESPECT compile-time option
WINDOW compile-time option
RULES compile-time option
Understanding date patterns
Patterns and windowing
Using built-in functions with MLE
DAYS
DAYSTODATE
Performing date calculations and comparisons
Explicit date calculations
Comparing dates
Converting dates
Subtracting dates
Implicit date calculations
Implicit date comparisons
Comparing dates with like patterns
Comparing dates with differing patterns
Comparisons involving the DATE attribute and a literal
Comparisons involving the DATE attribute and a non-literal
Implicit DATE assignments
Chapter 16. Using the Program Maintenance Utility, NMAKE

The Program Maintenance Utility (NMAKE) automates the process of updating project files. NMAKE compares the modification dates for one set of files (the target files) with those of another set of files (the dependent files). If any dependent files have changed more recently than the target files, NMAKE executes a series of commands to bring the targets up-to-date.

Why use NMAKE?

The most common use of NMAKE is to automate the process of updating a project after you make a change to a source file. Large projects tend to have many source files. Often, only a few of your source files need to be compiled when you make a change. You set up a special text file, called a description file (or makefile), that tells NMAKE:

- Which files depend on others
- Which commands, such as compile and link commands, need to be carried out to bring your program up-to-date.

This use of NMAKE is only one example of its power. By building suitable description files, you can use NMAKE to:

- Make backups
- Configure data files
- Run programs when data files are modified.

Running NMAKE

Run NMAKE by typing `nmake` on the operating-system command line. Supply input to NMAKE by either of two methods:

- Enter the input directly on the command line.
- Put your input into a `command file` (a text file, also called a `response file`) and enter the file name on the command line.

Press Ctrl+C at any time during an NMAKE run to return to the operating system.

Using the command line

When using NMAKE at the command line, keep the following in mind:

- All fields are optional.
- NMAKE always looks first in the current directory for a description file called `makefile`. If `makefile` does not exist, NMAKE uses the `filename` given with the `/F` (specify description file) option (see “Specify description file (/F)” on page 336).

Command-line syntax

```
nmake [options] [macrodefinitions] [targets] /F filename```

**options**

Specifies options that modify NMAKE’s actions.

**macrodefinitions**

Lists macro definitions for NMAKE to use. Macro definitions that contain spaces must be enclosed by double quotation marks.
Running NMAKE

targets
Specifies the names of one or more target files to build. If you do not list any targets, NMAKE builds the first target in the description file.

/F filename
Gives the name of the description file where you specify file dependencies and which commands to execute when a file is out-of-date.

The following example:
  nmake /s "program = flash" sort.exe search.exe
  - Invokes NMAKE with the /s option
  - Defines a macro, assigning the string "flash" to the macro "program"
  - Specifies two targets: sort.exe and search.exe

By default, NMAKE uses the file named makefile as the description file.

Command-line help
To display NMAKE help, type nmake /? at the prompt. The appropriate copyright statement appears, along with the following:

Usage:
  NMAKE @commandfile
  NMAKE /help
  NMAKE [/nologo] [/acdeinpqrst?] [/f makefile] [/x stderrfile]
  [macrodefs][targets]

What the options stand for
/a   Force all targets to be built
/c   Cryptic mode; suppress sign-on banner & warning messages
/d   Display modification dates
/e   Environment variables override macros in the makefile
/i   Ignore exit codes of commands invoked
/n   No execute mode; display commands only
/p   Print macro definitions & target descriptions
/q   Query if target is up to date; for use in batch files
/r   Inference Rules from 'TOOLS.INI' to be ignored
/s   Silent execution of commands
/t   Touch targets with current date & time
/?   Help message
/help Help message
/nologo Do not display sign-on banner

Using NMAKE command files
A command file is a response file used to extend command-line input to NMAKE.

You can split input to NMAKE between the command line and a command file. Use the name of a command file (preceded by @) where you normally type the input information on the command line.

Why use a command file?
Use a command file for:
  • Complex and long commands you type frequently
  • Strings of command-line arguments, such as macro definitions, that exceed the limit for command-line length.

A command file is not the same as a description file. For information about description files, see "Description files" on page 337.
Command file syntax
To provide input to NMAKE with a command file, type

```
nmake @commandfile
```

In the `commandfile` field, enter the name of a file containing the same information as is normally entered on the command line.

NMAKE treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines if you end each line except the last with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

Example
The following is a command file called `update`:

```
/s "program \n= flash" sort.exe search.exe
```

You can use this command file by typing the following command:

```
nmake @update
```

This runs NMAKE using:
- The `/s` option
- The macro definition "program = flash"
- The targets specified as `sort.exe` and `search.exe`
- The description file `makefile` by default

The backslash allows the macro definition to span two lines.

NMAKE options

You can use several options with NMAKE. Keep the following in mind when using options:
- Option characters are not case sensitive; `/I` and `/i` are equivalent.
- You can use either a slash or dash before the option characters; `-a` and `/a` are equivalent.

Produce error file (/X)
Syntax: `/X stderrfile`

This option produces a standard error file.

Build all targets (/A)
Syntax: `/A`

This option builds all specified targets even if they are not out-of-date with respect to their dependent files.

See "Description files" on page 337

Suppress messages (/C)
Syntax: `/C`
NMAKE options

This option suppresses display of the NMAKE sign-on banner, nonfatal error messages, and warning messages. To suppress the sign-on banner without suppressing other messages, use the /NOLOGO option.

Display modification dates (/D)

Syntax: /D

This option displays the modification date of each file when the dates of target and dependent files are checked.

See “Description files” on page 337.

Override environment variables (/E)

Syntax: /E

This option disables inherited macro redefinition.

NMAKE inherits all current environment variables as macros, which can be redefined in a description file. The /E option disables any redefinition — the inherited macro always has the value of the environment variable.

Specify description file (/F)

Syntax: /F filename

This option specifies filename as the name of the description file to use. If a dash (-) is entered instead of a file name, NMAKE reads a description file from the standard input device, typically the keyboard.

If a filename is not specified, it defaults to makefile.

Display help (/HELP or /?)

Syntax: /HELP or /?

This option displays a brief summary of NMAKE syntax.

Ignore exit codes (/I)

Syntax: /I

This option ignores exit codes (also called error level or return codes) returned by programs such as compilers or linkers called by NMAKE. If this option is not specified, NMAKE ends when any program returns a nonzero exit code.

Display commands (/N)

Syntax: /N

This option causes NMAKE commands to be displayed but not executed. Use the /N option to:
- Check which targets are out-of-date with respect to their dependents
- Debug description files

Suppress sign-on banner (/NOLOGO)

Syntax: /NOLOGO
NMAKE options

This option suppresses the sign-on banner display when NMAKE is started. If you want to suppress nonfatal error messages and warnings as well, use the suppress messages (/C) option.

Print macro and target definitions (/P)

Syntax: /P

This option writes out all macro definitions and target definitions. Output is sent to the standard output device (typically the display).

Return exit code (/Q)

Syntax: /Q

This option causes NMAKE to return either of the following:

- An exit code of zero if all targets built during an NMAKE run are up-to-date
- An exit code other than zero if they are not up-to-date

Use this option to run NMAKE from within a batch file.

Ignore TOOLS.INI file (/R)

Syntax: /R

This option ignores the following:

- All inference rules and macros contained in the TOOLS.INI file
- All predefined inference rules and macros

Suppress command display (/S)

Syntax: /S

This option suppresses the display of commands as they are executed by NMAKE. It does not suppress the display of messages generated by the commands themselves.

The /N command (Display Commands) takes precedence over the /S option. If you use /N and /S together, commands are displayed but not executed.

Change target modification dates (/T)

Syntax: /T

This option changes or “touches” the modification dates for out-of-date target files to the current date. No commands are executed, and the target file is left unchanged.

Description files

NMAKE uses a description file to determine what to do. In its simplest form, a description file tells NMAKE which files depend on others and which commands need to be executed if a file changes.

A description file looks like this:
Description files

```
targets...: dependents...
    command
    :

targets... : dependents...
    command
```

Description blocks
A dependent relationship between files is defined in a description block. A description block indicates the relationship among various parts of the program. It contains commands to bring all components up to date. The description file can contain up to 1048 description blocks.

Special features
The following are special features of description files and blocks:

- Description files can contain macro definitions and use macros in description blocks. Macros allow easy substitution of one text string for another.
- Description files can contain inference rules. Inference rules allow NMAKE to infer which commands to execute based on the filename extensions used for targets and dependents.
- You can specify directories for NMAKE to search for dependent files by using the following syntax:
  targets : {directory1;directory2...}dependents
  NMAKE searches the current directory first, then directory1, directory2, and so on.
- A command can be placed on the same line as the target and dependent files by using a semicolon (;) as depicted below:
  targets... : dependents... ; command
- A long command can span several lines if each line ends with a backslash ( \ ):
  command \
    continuation of command
- The execution of a command can be modified if you precede the command with special characters.
- If you do not specify a command in a description block, NMAKE looks for an inference rule to build the target.
- Wild card characters (* and ?) can be used in description blocks. For example, the following description block compiles all source files with the .PLI extension:
  astro.exe : *.pli
  pli $**
Description files

- NMAKE expands the *pli specification into the complete list of PL/I files in the current directory. $** is a complete list of dependents specified for the current target.
- NMAKE uses several punctuation characters in its syntax. To use one of these characters as a literal character, place an escape character (^) in front of it. For a list of punctuation characters, see "Escape characters" on page 351.
- Normally a target file can appear in only one description block. A special syntax allows you to use a target in several description blocks.
- A special syntax allows you to determine the drive, path, base name, and extension of the first dependent file in a description block.

Targets in several description blocks
Using a file as a target in more than one description block causes NMAKE to end. You can overcome this limitation by using two colons (::) as the target/dependent separator instead of one colon.

The following description block is permissible:

```
X :: A
   command
X :: B
   command
```

The following causes NMAKE to end:

```
X : A
   command
X : B
   command
```

It is permissible to use single colons if the target/dependent lines are grouped above the same commands. The following is permissible:

```
X : A
X : B
   command
```

Double colon (::) target/dependent separator example

```
target.lib :: a.asm b.asm c.asm
ml a.asm b.asm c.asm
lib target a.obj b.obj c.obj

target.lib :: d.pli e.pli
pli d.pli
pli e.pli
lib target d.obj e.obj
```

These two description blocks both update the library named target.lib. If any of the assembly-language files have changed more recently than the library file, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the PL/I language files have changed, NMAKE executes the second group of commands to compile the PL/I files and update the library.
Using macros

Macros provide a convenient way to replace one string with another in the description file. The text is automatically replaced each time NMAKE is run. This feature makes it easy to change text throughout the description file without having to edit every line that uses the text.

Two common uses of macros are:
- To create a standard description file for several projects. The macro represents the file names in commands. These file names are defined when you run NMAKE. When you switch to a different project, changing the macro changes the file names NMAKE uses throughout the description file.
- To control the options that NMAKE passes to the compiler, assembler, or linker. When using a macro to specify the options, you can quickly change the options throughout the description file in one easy step.

A macro can be defined:
- In a description file
- On the command line
- In TOOLS.INI
- Through inheritance from environment variables

Macros example

```plaintext
program = flash
c = ilink
options =

$(program).exe : $(program).obj
$c $(options) $(program).obj;
```

The example above defines three macros. The description block executes the following commands:

```
flash.exe : flash.obj
ilink flash.obj;
```

Special features

Macros have the following special features:
- When using a macro, you can substitute text in the macro itself.
- Several macros have been predefined for special purposes.
- If a macro is defined more than once, precedence rules govern which definition is used.
- You can also put macros into your TOOLS.INI file.

Macros in a description file

Before using a macro, you need to define it, either on the NMAKE command line or in your description file. Description file macro definitions look like this:

```plaintext
macroname = macrostring
```

Macro names can be any combination of alphanumeric characters and the underscore character (_), and they are case-sensitive. A macro string can be any string of characters.

The first character of the macro name must be the first character on the line. NMAKE ignores any spaces before or after the equal sign (=).
Using macros

The macro string can be a null string and can contain embedded spaces. Do not
enclose the macro string in quotation marks in the description file; quotation marks
are used only when you define macros on the command line.

Macros on the command line

Before using a macro, you need to define it, either on the NMAKE command line
or in your description file. Command-line macro definitions look like this:

macroname=macrostring

No spaces can surround the equal sign. If you embed spaces, NMAKE might
misinterpret your macro. If your macro string contains embedded spaces, enclose it
in double quotation marks ("') like this:

macroname="macro string"

You can also enclose the entire macro definition in double quotation marks ("') like
this:

"macroname = macro string"

Macro names can be any combination of alphanumeric characters and the
underscore character (_), and they are case-sensitive. A macro string can be any
string of characters or a null string.

Inherited macros

NMAKE inherits all current environment variables as macros. For example, if you
have a PATH environment variable defined as PATH = C:\TOOLS\BIN, the string
C:\TOOLS\BIN is substituted when you use PATH in the description file.

You can redefine inherited macros by including a line such as the example above
in a description file. While NMAKE is executing, the macro takes on the redefined
definition. When NMAKE terminates, however, the environment variable resumes
its original value.

The Override Environment Variables (/E) option disables inherited macro
redefinition. If you use this option, NMAKE ignores any attempt to redefine an
inherited macro.

The macro name, for any macros that you define, is case sensitive. For example,
consider this macro:

UPPER=UpperCase

In this example, $(UPPER) returns the value, but $(upper) does not. Inherited
macro names (i.e. those created automatically from environment variables) must
always be UPPERCASE.

Defined macros

After you have defined a macro, you can use it anywhere in your description file
with the following syntax:

$(macroname)

The parentheses are not required if the macro name is only one character long. To
use a dollar sign ($) without using a macro, enter two dollar signs ($$), or use the
caret (^) before the dollar sign as an escape character.
Using macros

When NMAKE runs, it replaces all occurrences of $(macroname) with the defined macro string. If the macro is undefined, nothing is substituted. After a macro is defined, you can cancel it only with the !UNDEF directive.

Macro substitutions

Just as you use macros to substitute text within a description file, you use the following syntax to substitute text within a macro:

```
$(macroname: string1 = string2)
```

Every occurrence of `string1` is replaced by `string2` in `macroname`. Spaces between the colon and `string1` are considered part of `string1`. If `string2` is a null string, all occurrences of `string1` are deleted from the macro. The colon (:) must immediately follow `macroname`.

The replacement of `string1` with `string2` in the macro is not a permanent change. If you use the macro again without a substitution, you get the original unchanged macro.

Example

```
SOURCES = one.pli two.pli three.pli
program.exe : $(SOURCES:.pli=.obj)
  ilink $**;
```

The example above defines a macro called SOURCES, which contains the names of three PL/I source files. With this macro, the target/dependent line substitutes the .obj extension for the .pli extension. Thus, NMAKE executes the following command:

```
  ilink one.obj two.obj three.obj;
```

$** is a special macro that translates to all dependent files for a given target.

Special macros

NMAKE predefines several macros. The first six macros below return one or more file specifications for the files in the target/dependent line of a description block. Except where noted, the file specification includes the path of the file, the base filename, and the filename extension.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$@</td>
<td>The specification of the target file.</td>
</tr>
<tr>
<td>$*</td>
<td>The base name (without extension) of the target file. Path information is also returned if the path was specified as part of the target filename. This macro cannot be used in a dependent list.</td>
</tr>
<tr>
<td>$**</td>
<td>The specifications of the dependent files.</td>
</tr>
<tr>
<td>$?</td>
<td>The specifications for only those dependent files that are out-of-date with respect to the targets.</td>
</tr>
<tr>
<td>$&lt;</td>
<td>The specification of a single dependent file that is out-of-date with respect to the targets. This macro is used only in inference rules.</td>
</tr>
<tr>
<td>$@AS</td>
<td>The file specification of the target that NMAKE is currently evaluating. This is a dynamic dependency parameter, used only in dependent lists.</td>
</tr>
<tr>
<td>$(AS)</td>
<td>The string MASM, which is the command to run the Macro Assembler (MASM). You can redefine this macro to use a different command.</td>
</tr>
</tbody>
</table>
Special macros

$(MAKE)
The command name used to run NMAKE. This macro is used to invoke NMAKE recursively. If you redefine this macro, NMAKE issues a warning message.

NMAKE executes the command line in which $(MAKE) appears, even if the display commands (/N) option is on.

$(MAKEFLAGS)
The NMAKE options currently in effect. You cannot redefine this macro.

The special macros $$ and $$ are the only exceptions to the rule that macro names longer than one character must be enclosed in parentheses.

You can append characters to any of the first six macros in this list to modify the meaning of the macro. However, you cannot use macro substitutions in these macros.

Special macros examples

trig.lib : sin.obj cos.obj arctan.obj
!ilib trig.lib $?

In this example, the macro $? represents the names of all dependent files that are out-of-date with respect to the target file. The exclamation point (!) preceding the ILIB command causes NMAKE to execute the ILIB command once for each dependent file in the list. As a result of this description, the ILIB command causes NMAKE to execute the ILIB command once for each dependent file in the list. As a result of this description, the ILIB command is executed up to three times, each time replacing a module with a newer version.

DIR=c:\include
$(DIR)\globals.inc : globals.inc
  copy globals.inc $@
$(DIR)\types.inc : types.inc
  copy types.inc $@
$(DIR)\macros.inc : macros.inc
  copy macros.inc $@

This example shows how to update a group of include files. Each of the files, globals.inc, types.inc, and macros.inc, in the directory c:\include depends on its counterpart in the current directory. If one of the include files is out-of-date, NMAKE replaces it with the file of the same name from the current directory.

The following description file, which uses the special macro $$, is equivalent:

DIR=c:\include
$(DIR)\globals.inc $(DIR)\types.inc $(DIR)\macros.inc : $$(@F)
!copy $? $@

The special macro $$(@F) signifies the file name (without the path) of the current target.

When NMAKE evaluates the description block, it evaluates the three targets, one at a time, with respect to their dependents. Thus, NMAKE first checks whether c:\include\globals.inc is out-of-date compared with globals.inc in the current directory. If so, it executes the command to copy the dependent file globals.inc to the target. NMAKE repeats the procedure for the other two targets.

Note that on the command line, the macro $? refers to the dependent for this target. The macro $@ specifies the full file specification of the target file.
Special macros

File-specification parts
A full file specification gives the base name of the file, the file-name extension, and the path. The path provides the disk-drive identifier and the sequence of directories needed to locate the file on the disk.

For example, the file specification
\[ c:\source\prog\sort.obj \]

has the following parts:
- **Path Name**: \[ c:\source\prog \]
- **Base File Name**: sort
- **File-Name Extension**: .obj

Characters that modify special macros
The following six macros all resolve to a file specification (or possibly several file specifications for \$** and \$?):

\[ \$* \quad \$@ \quad \$** \quad \$< \quad \$? \quad $$$@ \]

You can append characters to any of these macros to modify the file name returned by the macro. Depending on which character you use, parts of the full file specification are returned:

<table>
<thead>
<tr>
<th>Appended Character</th>
<th>File Part Returned</th>
<th>D</th>
<th>F</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Path</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Base File Name</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>File Name Extension</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Modified special macros example
If the macro \$@ has the value
\[ c:\source\prog\sort.obj \]

then the following values are returned for the modified macro:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(@D)</td>
<td>c:\source\prog</td>
</tr>
<tr>
<td>$(@F)</td>
<td>sort.obj</td>
</tr>
<tr>
<td>$(@B)</td>
<td>sort</td>
</tr>
<tr>
<td>$(@R)</td>
<td>c:\source\prog\sort</td>
</tr>
</tbody>
</table>

Modified macros are always longer than a single character — they must be enclosed by parentheses when used.

Macro precedence rules
When the same macro is defined in more than one place, the definition with the highest priority is used:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Highest)</td>
<td>Command line</td>
</tr>
<tr>
<td>2</td>
<td>Description file</td>
</tr>
<tr>
<td>3</td>
<td>Environment variables</td>
</tr>
</tbody>
</table>
Inference rules

Inference rules are templates from which NMAKE infers what to do with a
description block when no commands are given. Only those extensions defined in
a .SUFFIXES list can have inference rules. The extensions .c, .obj, .asm, and .exe are
automatically included in .SUFFIXES.

PL/I programmers

You must add PL/I file extensions manually using the .SUFFIXES pseudotarget.
See "SUFFIXES Pseudotarget" on page 350.

When NMAKE encounters a description block with no commands, it looks for an
inference rule that specifies how to create the target from the dependent files,
given the two file extensions. Similarly, if a dependent file does not exist, NMAKE
looks for an inference rule that specifies how to create the dependent from another
file with the same base name.

NMAKE applies an inference rule only if the base name of the file it is trying to
create matches the base name of a file that already exists.

In effect, inference rules are useful only when there is a one-to-one correspondence
between the files with the "from" extension and the files with the "to" extension.
You cannot, for example, define an inference rule that inserts a number of modules
into a library.

The use of inference rules eliminates the need to put the same commands in
several description blocks. For example, you can use inference rules to specify a
single pli command that changes any PL/I source file (with a .pli extension) to an
object file (with a .obj extension).

You define an inference rule by including text of the following form in your
description file or in your TOOLS.INI file — see “Special Features”.

```
.fromext.toext:
  commands

The elements of the inference rule are:
fromext
  The file-name extension for dependent files to build a target
toext
  The file-name extension for target files to be built
commands
  The commands to build the toext target from the fromext dependent.
```

For example, an inference rule to convert PL/I source files (with the .pli
extension) to PL/I object files (with the .obj extension) is
Inference rules

\begin{verbatim}
.pli.obj:
 pli $<

The special macro $< represents the name of a dependent out-of-date relative to the target.
\end{verbatim}

Special features

- You can specify a path where NMAKE should look for target and dependent files used in inference rules.
- Inference rules are predefined for compiling and linking C programs, and for assembling programs.
- NMAKE looks for inference rules in the T00LS.INI file if it cannot find a rule in a description file.
- Only those extensions defined in a .SUFFIXES list can have inference rules. The extensions .c, .obj, .asm, and .exe are automatically included in .SUFFIXES.
- You must add PL/I file extensions manually using the .SUFFIXES pseudotarget. See “.SUFFIXES Pseudotarget” on page 350.

Inference rules example

\begin{verbatim}
.obj.exe:
 ilink $<

.example1.exe: example1.obj

.example2.exe: example2.obj
 ilink /co example2,,,libv3.lib
\end{verbatim}

The first line above defines an inference rule that causes the ILINK command to create an executable file whenever a change is made in the corresponding object file. The file name in the inference rule is specified with the special macro $< so that the rule applies to any .obj file with an out-of-date executable file.

When NMAKE does not find any commands in the first description block, it checks for a rule that might apply and finds the rule defined on the first two lines of the description file. NMAKE applies the rule, replacing $< with example1.obj when it executes the command, so that the ILINK command becomes

\begin{verbatim}
ilink example1.obj;
\end{verbatim}

NMAKE does not search for an inference rule when examining the second description block, because a command is explicitly given.

Inference-rule path specifications

When defining an inference rule, you can indicate to NMAKE where to look for target and dependent files. Use the following syntax:

\begin{verbatim}
{frompath}.fromext{topath}.toext
 commands
\end{verbatim}

NMAKE looks in the directory specified by frompath for files with the fromext extension. It executes the commands to build files with the toext extension in the directory specified by topath.
Inference rules

Predefined inference rules

NMAKE predefines several inference rules:

<table>
<thead>
<tr>
<th>Inference Rule</th>
<th>Command Action</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c.obj</td>
<td>$(CC) $(CFLAGS) /c $*.c</td>
<td>icc /c $*.c</td>
</tr>
<tr>
<td>.c.exe</td>
<td>$(CC) $(CFLAGS) $*.c</td>
<td>icc $*.c</td>
</tr>
<tr>
<td>.asm.obj</td>
<td>$(AS) $(AFLAGS) $*;</td>
<td>masm $*;</td>
</tr>
</tbody>
</table>

- The first two rules automatically compile and link C programs.
- The last rule automatically assembles programs.
- The above are the most often used predefined inference rules. For a complete list of predefined inference rules, execute a makefile and specify the /p option. All available inference rules will be displayed.

Directives

Using directives, you can construct description files similar to batch files. NMAKE provides directives that:
- Conditionally execute commands
- Display error messages
- Include the contents of other files
- Turn some NMAKE options on or off

Each directive begins with an exclamation point (!) in the first column of the description file. Spaces can be placed between the exclamation point and the directive keyword.

The list below describes the directives:

**!IF expression**

Executes the statements between the !IF keyword and the next !ELSE or !ENDIF directive if expression evaluates to a nonzero value.

The expression used with the !IF directive can consist of integer constants, string constants, or exit codes returned by programs. Integer constants can use the C unary operators for numerical negation ( - ), one's complement ( ~ ), and logical negation ( ! ). You can also use any of the C binary operators listed below:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^^</td>
<td>Bitwise XOR</td>
</tr>
</tbody>
</table>
Directives

&& Logical AND
|| Logical OR
<< Left shift
>> Right shift
== Equality
!= Inequality
< Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to

- You can use parentheses to group expressions.
- Values are assumed to be decimal values unless specified with a leading 0 (octal) or leading 0x (hexadecimal).
- Strings are enclosed by quotation marks (" "). You can use the equality (==) and inequality (!=) operators to compare two strings.
- You can invoke a program in an expression by enclosing the program name in square brackets ([ ]). The exit code returned by the program is used in the expression.

!ELSE Executes the statements between the !ELSE and !ENDIF directives if the statements preceding the !ELSE directive were not executed.

!ENDIF Marks the end of the !IF, !IFDEF, or !IFNDEF block of statements.

!IFDEF macroname
Executes the statements between the !IFDEF keyword and the next !ELSE or !ENDIF directive if macroname is defined in the description file. If a macro has been defined as null, it is still considered to be defined.

!IFNDEF macroname
Executes the statements between the !IFNDEF keyword and the next !ELSE or !ENDIF directive if macroname is not defined in the description file.

!UNDEF macroname
Undefines a previously defined macro.

!ERROR text
Prints text and then stops execution.

!INCLUDE filename
Reads and evaluates the file filename before continuing with the current description file. If filename is enclosed by angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise, it looks only in the current directory. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.

!CMDSWITCHES {+1-}opt
Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If no options are specified, the options are reset to the values they had when NMAKE was started. To turn an option on, precede it with a plus sign (+); to turn it off, precede it with a minus sign (-). This directive updates the MAKEFLAGS macro.

See “Special macros” on page 342.
Directives example

!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe:winner.obj
!IFDEF DEBUG
! IF "$(DEBUG)"=="y"
   ilink /de winner.obj;
! ELSE
   ilink winner.obj;
! ENDIF
!ELSE
! ERROR Macro named DEBUG is not defined.
!ENDIF

The directives in this example do the following:
- The !INCLUDE directive causes the file infrules.txt to be read and evaluated as if it were part of the description file.
- The !CMDSWITCHES directive turns on the /D option, which displays the dates of the files as they are checked.
- If winner.exe is out-of-date with respect to winner.obj, the !IFDEF directive checks to see whether the macro DEBUG is defined. If it is defined, the !IF directive checks to see whether it is set to y. If it is, the linker is invoked with the /DE option; otherwise, it is invoked without the /DE. If the DEBUG macro is not defined, the !ERROR directive prints the message and NMAKE stops executing.

Pseudotargets

A pseudotarget is a target in a description block that is not a file. Instead, it is a name that serves as a "handle" for building a group of files or executing a group of commands. In the following example, UPDATE is a pseudotarget:

UPDATE: *.*
!copy $** a:\product

When NMAKE evaluates a pseudotarget, it always considers the dependents to be out-of-date. In the description above, NMAKE copies each of the dependent files to the specified drive and directory.

NMAKE predefines several pseudotargets for special purposes.

See "Predefined pseudotargets."

Predefined pseudotargets

NMAKE predefines several pseudotargets that provide special rules within a description file:

.SILENT Pseudotarget

Syntax: .SILENT : dependents...

This pseudotarget suppresses the display of executed commands for a single description block. The /S option does the same thing for all description blocks.

See "Suppress command display (/S)" on page 337.

.IGNORE Pseudotarget

Syntax: .IGNORE : dependents...


Directives

This pseudotarget ignores exit codes returned by programs for a single description block. The /I option does the same thing for all description blocks.

See "Ignore exit codes (/I)" on page 336.

.SUFFIXES Pseudotarget

Syntax: .SUFFIXES : extensions...

This pseudotarget defines file extensions to try when NMAKE needs to build a target file for which no dependents are specified. NMAKE searches the current directory for a file with the same name as the target file and an extension in <extensions...>. If NMAKE finds such a file, and if an inference rule applies to the file, NMAKE treats the file as a dependent of the target.

The .SUFFIXES pseudotarget is predefined as

.SUFFIXES : .obj .exe .c .asm

To add extensions to the list, specify .SUFFIXES : followed by the new extensions. For example, the following would enable you to write inference rules for PL/I source files.

.SUFFIXES: .pli

To clear the list, specify

.SUFFIXES:

Only those extensions specified in .SUFFIXES can have inference rules. NMAKE ignores inference rules unless the extensions have been specified in a .SUFFIXES list.

.PRECIOUS Pseudotarget

Syntax: .PRECIOUS : targets...

This pseudotarget tells NMAKE not to delete a target even if the commands that build it are terminated or interrupted. This pseudotarget overrides the NMAKE default. By default, NMAKE deletes the target if it cannot be sure that the target was built successfully.

For example,

.PRECIOUS : tools.lib

If the commands to build tools.lib are interrupted, leaving an incomplete file, NMAKE does not delete the partially built tools.lib.

The pseudotarget .PRECIOUS is useful only in limited circumstances. Most professional development tools have their own interrupt handlers and "clean up" when errors occur.

Inline files

You may need to issue a command in the description file with a list of arguments exceeding the command-line limit of the operating system. Just as NMAKE supports the use of command files, it can also generate inline files which are read as response files by other programs.
**Inline files**

To generate an inline file, use the following syntax for your description block:

```
target : dependents
  command @<<[filename]
inline file text
<< [KEEP | NOKEEP]
```

All of the text between the two sets of double less than signs (<<) is placed into an inline file and given the name `filename`. You can refer to the inline file at a later time by using `filename`. If `filename` is not given, NMAKE gives the file a unique name in the directory specified by the TMP environment variable if it is defined. Otherwise, NMAKE creates a unique file name in the current directory.

The inline file can be temporary or permanent. If you do not specify otherwise, or if you specify the keyword NOKEEP, the inline file is temporary. Specify KEEP to retain the file.

The at sign (@) is not part of the NMAKE syntax but is the typical character used by utilities to designate a file as a response file.

**Inline files example**

```
math.lib : add.obj sub.obj mul.obj div.obj
    ilib @<<
math.lib
add.obj sub.obj mul.obj div.obj
/L:listing
<<
```

The above example creates an inline file and uses it to invoke the Library Manager (ILIB). The inline file is used as a response file by (ILIB). It specifies which library to use, the commands to execute, and the listing file to produce. The inline file contains the following:

```
math.lib
add.obj sub.obj mul.obj div.obj
/L:listing
```

Because no file name is listed after the ILIB command, the inline file is given a unique name and placed into the current directory (or the directory defined by the TMP environment variable).

**Escape characters**

NMAKE uses the following punctuation characters in its syntax:

<table>
<thead>
<tr>
<th>(</th>
<th>)</th>
<th>#</th>
<th>$</th>
<th>^</th>
<th>\</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>]</td>
<td>!</td>
<td>@</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

To use one of these characters in a command and not have it interpreted by NMAKE, use a caret (^) in front of the character.

For example,

```
BIG^#.PLI
```

is treated as

```
BIG#.PLI
```
With the caret, you can include a literal newline character in a description file. This capability is useful in macro definitions, as in the following example:

```
XYZ=abc\<ENTER>
def
```

The effect is equivalent to the effect of assigning the C-style string `abc\ndef` to the `XYZ` macro. Note that this effect differs from the effect of using the backslash (\) to continue a line. A newline character that follows a backslash is replaced with a space.

NMAKE ignores a caret that is not followed by any of the characters it uses in its syntax. A caret that appears within quotation marks is not treated as an escape character.

The escape character cannot be used in the command portion of a dependency block.

### Characters that modify commands

Any of three characters can be placed in front of a command to modify how the command is run:

— (dash)
  
  Turns off error checking for the command

@ (at sign)
  
  Suppresses display of the command

! (exclamation point)
  
  Executes the command for each dependent file

Spaces can separate the modifying character from the command. Any command on a separate line — whether modified or not — must be indented by one or more spaces or tabs.

You can use more than one character to modify a single command.

#### Turn error checking off (-)

**Syntax:** `-n` command

The `/I` option globally turns command error-checking off. The dash (-) command modifier overrides the global setting to turn error checking off for commands individually. This modifier is used in two ways:

- A dash without a number turns off all error checking.
- A dash followed by a number causes NMAKE to abort only if the exit code returned by the command is greater than the number.

See “Ignore exit codes (/I)” on page 336.

### Dash command modifier examples

```
ligh.lst : light.txt
- flash light.txt
```

In the example above, NMAKE never ends, regardless of the exit code returned by `flash`. 
Characters that modify commands

light.lst : light.txt
    -1 flash light.txt

In the example above, NMAKE ends if the exit code returned by flash is greater than 1.

Suppress command display (@)

**Syntax:** @ command

The /S option globally suppresses the display of commands while NMAKE is running. The at sign (@) modifier suppresses the display for individual commands.

Regardless of the /S option or the @ modifier, output generated by the command itself always appears.

See “Suppress command display (/S)” on page 337.

At sign (@) command modifier example

Suppress Command Display (@)

sort.exe:sort.obj
    @ echo sorting

The command line calling the echo command is not displayed. The output of the echo command, however, is displayed.

Execute command for dependents (!)

**Syntax:** ! command

The exclamation-point command modifier causes the command to be executed for each dependent file if the command uses one of the special macros $? or $**. The $? macro refers to all dependent files out-of-date with respect to the target. The $** macro refers to all dependent files in the description block.

See “Special macros” on page 342.

Exclamation point (!) command modifier examples

leap.txt : hop.asm skip.c jump.pli
    ! print $** lpt1:

The example above executes the following three commands, regardless of the modification dates of the dependent file:

    print hop.asm lpt1:
    print skip.c lpt1:
    print jump.pli lpt1:

leap.txt : hop.asm skip.c jump.pli
    ! print $? lpt1:

The example above executes the print command only for those dependent files with modification dates later than that of the leap.txt file. If hop.asm and jump.pli have modification dates later than leap.txt, the following two commands are executed:

    print hop.asm lpt1:
    print jump.pli lpt1:
EXTMAKE Syntax

Description files can use a special syntax to determine the drive, path, base name, and extension of the first dependent file in a description block. This syntax is called the extmake syntax.

The characters %s represent the complete file specification of the first dependent file. Various parts of the file specification are represented using the following syntax:

```
%<parts>
```

- `<parts>`
  - A combination of the following letters:
    - d Drive
    - p Path
    - f Base name
    - e Extension

For example, to specify the drive and path name of the first dependent file in a description block, use:
```
%<dp>
```

The percent symbol (%) is a replacement in DOS and Windows command lines. To use the percent symbol in command-line arguments, use a double percent (%%).

Macros and inference rules in TOOLS.INI

You can place either macros or inference rules in your TOOLS.INI file. NMAKE looks for the TOOLS.INI file first in the current directory and then in the directory indicated by the INIT environment variable.

If NMAKE finds a TOOLS.INI file, it looks for the following tag:
```
[nmake]
```

You can place macros and inference rules below this tag in the same format you would use in a description file.

If a macro or inference rule is defined in both the TOOLS.INI file and the description file, the definition in the description file takes precedence. Also, if you use the /R option, the TOOLS.INI file is ignored.

TOOLS.INI example

```
[nmake]
.SUFFIXES: .pli
COMPILE_OPTS = gonumber source
.pli.obj:
  PLI $*.pli ($(COMPILE_OPTS)
```

These lines in the TOOLS.INI file do the following:

- Add the .pli file extension to the list of extensions that can have inference rules.
- Define the COMPILE_OPTS macro as gonumber source.
- Define an inference rule to build .obj files from .pli source files.
Chapter 17. 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 workstation PL/I compiler and the code it generates.

Selecting compile-time options for optimal performance

The compile-time 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 compile-time options without editing your source code because these options can be specified on the command line or in the environment variable IBM.OPTIONS.

If you want to avoid details, the least complex way to improve the performance of generated code is to specify the following (non-default) compile-time options:

- PREFIX(NOFOFL)
- IMPRECISE
- OPT(2)
- DFT(REORDER)

The first two options can affect the semantics of your program, but generally only do so in unusual situations. If you specify the first two options, your code is improved even when compiled with optimization turned off. By using these options, the compiler is also less likely to make errors.

The following sections describe, in more detail, performance improvements and trade-offs associated with specific compile-time 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 can occur, 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 in this case is usually offset by the execution of fewer instructions in other places.

IMPRECISE

When you select this option, the compiler generates smaller and faster sequences of instructions for floating-point operations. This can have a significant effect on the performance of programs that contain floating-point expressions, either separately or in loops.

However, when programs are compiled with the IMPRECISE option, floating-point exceptions might not be reported at the precise location where they occur. (This is
especially true when the OPTIMIZE option is in effect.) In addition, floating-point operations can produce results that are not precisely IEEE conforming.

**GONUMBER**

Using this option results in a statement number table used for debugging. This added information can be extremely helpful when debugging, but including statement number tables increases the size of your executable file. Larger executable files can take longer to load.

By using one of the linker options, you can include the statement number tables in your executable during development to help with debugging. The /DEBUG (/DE) option directs the linker to include these tables in the executable file, so by not specifying /DE with the ILINK command, you can better control the size of your executable files. If the size of your executable file is a consideration, you can leave the tables out during production mode.

**SNAP**

When you use the SNAP option, the compiler generates extra instructions in the prolog and epilog code for every block. These instructions ensure that the run-time traceback messages (produced by PLIDUMP and the SNAP option on an ON statement) include all procedures that were active when the traceback was requested.

A trade-off of using the SNAP option and creating these additional instructions is that it can have a negative impact on the performance of your application. This is especially true for procedures that are called frequently.

**RULES**

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:

```pli

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.

When you use the RULES(LAXCTL) option, the compiler allows you to declare a CONTROLLED variable with a constant extent and then ALLOCATE it with a different extent, as in
DECLARE X BIT(1) CTL;
ALLOCATE X BIT(63);

However, this programming practice forces the compiler to assume that no CONTROLLED variable has constant extents, and consequently it will generate much less efficient code when these variables are referenced.

But, if you specify a constant extent for a CONTROLLED variable only when it will always have that length (or bound), then you will get much better performance if you specify the option RULES(NOLAXCTL).

PREFIX
This option determines if 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. The default suboptions are:

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. With personal computers, however, the hardware does not raise this condition so the compiler must generate extra code. This extra code can negatively impact the performance of your program; and unless your program requires (or expects) this condition to be raised, specify PREFIX(NOFIXEDOVERFLOW) to improve performance.

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. The default suboptions are:

IBM
BYADDR
RETURNS(BYVALUE)
The IBM/ANS, ASSIGNABLE/NONASSIGNABLE, and DIRECTED/ NODIRECTED suboptions have no effect on program performance. All of the other suboptions can affect performance to varying degrees and, if applied inappropriately, can make your program invalid.

**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. However, passing a variable by reference can hinder performance in two ways:

1. Every reference to that parameter requires an extra instruction.
2. Since 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.

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**

```pli
a: proc( parm1, parm2, ..., parmN );
   
   dcl parm1 byaddr ...;
   dcl parm2 byvalue ...;
   
   dcl parmN byvalue ...;
   /* program logic */

end;
```
Faster, equivalent function with BYVALUE parameter

```haskell
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.

**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.

**(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.

**(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 optimizations on such variables. Consequently, if your program logic allows, use DEFAULT(REORDER) to generate superior code.

**LINKAGE**
This suboption tells the compiler the default linkage to 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.

**ASCII or EBCDIC**
The DEFAULT(ASCII) option indicates that, by default, character data is held in native Intel style. When you specify the EBCDIC suboption, the compiler must generate extra instructions for most operations involving the input or output of character variables.

**IEEE or HEXADEC**
The DEFAULT(IEEE) option indicates that, by default, float data is to be held in native Intel style. When you specify the HEXADEC suboption, the compiler must execute significantly more instructions for most operations involving floating-point variables.

**(NON)NATIVE**
The DEFAULT(NATIVE) option indicates that, by default, fixed binary data, offset data, ordinal data, and the length prefix of varying strings are held in native Intel style. When you specify NONNATIVE, extra instructions are generated for operations involving those data types previously listed.

**(NO)INLINE**
The suboption NOINLINE 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:
- 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.
Improving performance

Summary of compile-time options that improve performance

In summary, the following options (if appropriate for your application) can improve performance:

- OPTIMIZE(2)
- IMPRECISE
- NOSNAP
- PREFIX(NOFIXEDOVERFLOW)
- RULES(ANS NOLAXCTL)
- DEFAULT with the following suboptions
  - BYVALUE
  - RETURNS(BYVALUE)
  - CONNECTED
  - NODESCRIPTOR
  - REORDER
  - ASCII
  - IEEE
  - NATIVE
  - LINKAGE(OPTLINK)

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 sections 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:

```plaintext
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 which 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 dosScanEnv is an input-only CHAR VARYINGZ string:

```pli
dcl dosScanEnv entry( char(*) varyingz nonasgn byaddr,
               pointer byaddr )
  returns( native fixed bin(31) optional )
  options( nodelcriptor linkage(system) );
```

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.

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
• The source string is truncated if it is longer than the target.

This assurance comes at the price 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 example which follows, faster code is generated for the second assignment statement:

```pli
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 your loop control variables are one of the types in the following list. You should rarely, if ever, use other types of variables.

• FIXED BINARY with zero scale factor
• FLOAT
• ORDINAL
• HANDLE
Coding for better performance

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.

Performance is decreased if your program depends not only on the value of a loop control variable, but also on its address. For example, if the ADDR built-in function is applied to the variable or if the variable is passed 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

```plaintext
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

```plaintext
p: package exports( a );
   dcl ib based fixed bin;

a: proc;
   dcl (i,j,k) fixed bin;
   ...
   ...
   call b( addr(i) );
   ...
   ...
```
Coding for better performance

```pli
b: proc( px );
   dcl px pointer;
   display( px->ib );
end;
end p;

REDUCIBLE functions

REDUCIBLE indicates that a procedure or 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 since REDUCIBLE is part of the declaration. If IRREDUCIBLE had been used in the declaration, \( f \) would be invoked twice.

```pli
dcl (f) entry options( reducible ) returns( fixed bin );
select;
   when( f(x)<0 )
     .
   when( f(x)>0 )
     .
   otherwise
     .
end;
```

DEFINED versus UNION

The UNION attribute is more powerful than the DEFINED attribute and provides more function. 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.

```pli
dcl b1 bit(31);
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 since 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 using the VALUE attribute. However, 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 bit(8n) where n is a constant.

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 using a library routine. The conversions done with code generated inline are shown in Table 25.

Table 25. Conditions under which conversions are handled inline

<table>
<thead>
<tr>
<th>Target</th>
<th>Source</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed bin(p1,q1)</td>
<td>fixed bin(p2,q2)</td>
<td>always</td>
</tr>
<tr>
<td>float(p2)</td>
<td></td>
<td>if SIZE is disabled</td>
</tr>
<tr>
<td>bit(1)</td>
<td></td>
<td>always</td>
</tr>
<tr>
<td>bit(n) aligned</td>
<td></td>
<td>if n is known and n ≤ 31</td>
</tr>
<tr>
<td>char(1)</td>
<td></td>
<td>if CONV is disabled</td>
</tr>
<tr>
<td>pic'(n)9'</td>
<td></td>
<td>if n ≤ 6</td>
</tr>
<tr>
<td>pic'(n)Z(m)9'</td>
<td></td>
<td>if n + m ≤ 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fixed dec(p1,q1)</th>
<th>fixed dec(p2,q2)</th>
<th>done using an especially fast library routine</th>
</tr>
</thead>
</table>


Coding for better performance
Table 25. Conditions under which conversions are handled inline (continued)

<table>
<thead>
<tr>
<th>Target</th>
<th>Source</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>float(p1)</td>
<td>fixed bin(p2,q2)</td>
<td>always</td>
</tr>
<tr>
<td>float(p2)</td>
<td></td>
<td>always</td>
</tr>
<tr>
<td>bit(1)</td>
<td></td>
<td>always</td>
</tr>
<tr>
<td>bit(n) aligned</td>
<td></td>
<td>if n is known and n ≤ 31</td>
</tr>
<tr>
<td>char(1)</td>
<td></td>
<td>if CONV is disabled</td>
</tr>
<tr>
<td>pic'(n)9'</td>
<td></td>
<td>if n ≤ 6</td>
</tr>
<tr>
<td>pic'(n)Z(m)9'</td>
<td></td>
<td>if n ≤ 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pictured fixed</td>
<td>pictured fixed</td>
<td>if pictures match</td>
</tr>
<tr>
<td>pictured float</td>
<td>pictured float</td>
<td>if pictures match</td>
</tr>
<tr>
<td>char</td>
<td>char nonvarying</td>
<td>always</td>
</tr>
<tr>
<td></td>
<td>char varying</td>
<td>always</td>
</tr>
<tr>
<td></td>
<td>char varyingz</td>
<td>always</td>
</tr>
<tr>
<td></td>
<td>pictured fixed</td>
<td>always</td>
</tr>
<tr>
<td></td>
<td>pictured float</td>
<td>always</td>
</tr>
<tr>
<td></td>
<td>pictured char</td>
<td>always</td>
</tr>
<tr>
<td>pictured char</td>
<td>pictured char</td>
<td>if pictures match</td>
</tr>
<tr>
<td>bit(1) nonvarying</td>
<td>bit(1) nonvarying</td>
<td>always</td>
</tr>
<tr>
<td>bit(n) nonvarying</td>
<td>bit(m) nonvarying</td>
<td>see note</td>
</tr>
</tbody>
</table>

**Note:** If all of the following apply:
1) source and target are byte-aligned
2) n and m are known
3) mod(m,8)=0 or n=m or source is a constant
4) mod(n,8)=0 or target is a scalar with STATIC, AUTOMATIC, or CONTROLLED attributes

Many string-handling built-in functions are evaluated through calls to library routines, but some are handled without a library call. Table 26 lists these built-in functions and the conditions under which they are handled inline.

Table 26. Conditions under which string built-in functions are handled inline

<table>
<thead>
<tr>
<th>String function</th>
<th>Comments and conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>When the third argument is a constant. The first two arguments must also be either both bit(1) or both aligned bit(n) where n is 8, 16 or 32. The function is also handled inline if it can be reduced to a bitwise infix operation and both arguments are aligned bit.</td>
</tr>
<tr>
<td>COPY</td>
<td>When the first argument has type character.</td>
</tr>
<tr>
<td>EDIT</td>
<td>When the first argument is REAL FIXED BIN, the SIZE condition is disabled, and the second argument is a constant string consisting of all 9's.</td>
</tr>
<tr>
<td>HIGH</td>
<td>Always</td>
</tr>
<tr>
<td>INDEX</td>
<td>When only two arguments are supplied and they have type character.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Always</td>
</tr>
<tr>
<td>LOW</td>
<td>Always</td>
</tr>
<tr>
<td>MAXLENGTH</td>
<td>Always</td>
</tr>
<tr>
<td>SEARCH</td>
<td>When only two arguments are supplied and they have type character.</td>
</tr>
</tbody>
</table>
Table 26. Conditions under which string built-in functions are handled inline (continued)

<table>
<thead>
<tr>
<th>String function</th>
<th>Comments and conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCHR</td>
<td>When only two arguments are supplied and they have type character.</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>When STRINGRANGE is disabled.</td>
</tr>
<tr>
<td>TRANSLATE</td>
<td>When the second and third arguments are constant.</td>
</tr>
<tr>
<td>TRIM</td>
<td>When only one argument is supplied and it has type character.</td>
</tr>
<tr>
<td>UNSPEC</td>
<td>Always</td>
</tr>
<tr>
<td>VERIFY</td>
<td>When only two arguments are supplied and they have type character.</td>
</tr>
<tr>
<td>VERIFYR</td>
<td>When only two arguments are supplied and they have type character.</td>
</tr>
</tbody>
</table>
Chapter 18. Using user exits

PL/I provides a number of user exits that allow you to customize the PL/I product to suit your needs. The workstation PL/I and PL/I for AIX 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, we recommend that you modify the supplied source files as appropriate.

The types of files provided include:
- PL/I source files with the extension PLI that are located in ..\samples.
- PL/I include files with the extension CPY that are located in ..\include. When compiling the user exits, make sure to set the INCLUDE or IBM.SYSLIB environment variables so that the CPY files can be found.
- Linker definition files with the extension DEF that are located in ..\samples.
- Control files (if applicable to the exit) with the extension INF that are located in ..\samples. When using the user exits, make sure the directory containing the INF files is specified using the appropriate environment variables (usually DPATH).

Using the compiler user exit

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 functions. With PL/I, you can write your own user exit or use the exit provided with the product, either 'as is' or slightly modified depending on what you want to do with it. The purpose of this chapter is to describe:
- 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
- Termination

As illustrated in [Figure 28 on page 370], 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.
Using the compiler user exit

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. For specific information on the global control block, see “Structure of global control blocks” on page 372.

- 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 373, “Writing the message filtering procedure” on page 373, and “Writing the termination procedure” on page 375.

### Activating the compiler user exit

In order to activate the compiler user exit, you must specify the EXIT compile-time option. For more information on the EXIT option, see “EXIT” on page 57.

The EXIT compile-time option allows you to specify a user-option-string which specifies the message control file. If you do not specify a string, IBMUEXIT.INF is used (see “Modifying IBMUEXIT.INF” on page 371), but you have to tell the computer where to find it. The default behavior, provided you do not change the IBMUEXIT.PLI sample program, is that the compiler looks for IBMUEXIT.INF in the current directory first and then in the directories specified in DPATH.

The user-option-string is passed to the user exit functions in the global control block which is discussed in “Structure of global control blocks” on page 372. Please refer to the field “Uex.UIB_User_char_str” in the section “Structure of global control blocks” on page 372 for additional information.

### The IBM-supplied compiler exit, IBMUEXIT

IBM supplies you with the sample compiler user exit, IBMUEXIT, which filters messages for you. It monitors messages and, based on the message number that you specify, suppresses the message or changes the severity of the message.

There are several files that comprise IBMUEXIT:

**IBMUEXIT.PLI**
Contains the PL/I source code.

**IBMUEXIT.DLL**
Executable DLL of IBMUEXIT.PLI. In order to build this file, issue the following commands from the command line:
Using the compiler user exit

On Windows:
pli ibmexit
ilib /geni ibmexit.def
ilink /dll ibmexit.obj ibmexit.exp

IBMUEXIT.DEF
DEF file that is used to build IBMUEXIT.DLL.

IBMUEXIT.INF
Control file that specifies filtering of messages.

The PLI source file is provided for your information and modification. The INF control file contains the message numbers that should be monitored, and tells IBMUEXIT what actions to take for them. The executable module reads the INF control file, and either ignores the message or changes its severity.

Customizing the compiler user exit
As was mentioned earlier, you can write your own compiler user exit or simply modify IBMUEXIT.PLI. In either case, the name of the executable file for the compiler user exit must be IBMUEXIT.DLL.

This section describes how to:
• Modify IBMUEXIT.INF for customized message filtering
• Create your own compiler user exit

Modifying IBMUEXIT.INF
Rather than spending the time to write a completely new compiler user exit, you can modify the sample program, IBMUEXIT.INF.

Edit the INF file to indicate which message numbers you want to suppress, and which message number severity levels you would like changed. A sample IBMUEXIT.INF file is shown in Figure 29.

<table>
<thead>
<tr>
<th>Fac Id</th>
<th>Msg No</th>
<th>Severity</th>
<th>Suppress</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>'IBM'</td>
<td>1041</td>
<td>-1</td>
<td>1</td>
<td>Comment spans multiple lines</td>
</tr>
<tr>
<td>'IBM'</td>
<td>1044</td>
<td>-1</td>
<td>1</td>
<td>FIXED BIN 7 mapped to 1 byte</td>
</tr>
<tr>
<td>'IBM'</td>
<td>1172</td>
<td>0</td>
<td>0</td>
<td>Select without OTHERWISE</td>
</tr>
<tr>
<td>'IBM'</td>
<td>1052</td>
<td>-1</td>
<td>1</td>
<td>Nodescriptor with * extent args</td>
</tr>
<tr>
<td>'IBM'</td>
<td>1047</td>
<td>12</td>
<td>0</td>
<td>Reorder inhibits optimization</td>
</tr>
<tr>
<td>'IBM'</td>
<td>8009</td>
<td>-1</td>
<td>1</td>
<td>Semicolon in string constant</td>
</tr>
<tr>
<td>'IBM'</td>
<td>1107</td>
<td>12</td>
<td>0</td>
<td>Undeclared ENTRY</td>
</tr>
<tr>
<td>'IBM'</td>
<td>1169</td>
<td>0</td>
<td>1</td>
<td>Precision of result determined by arg</td>
</tr>
</tbody>
</table>

Figure 29. Example of an IBMUEXIT.INF 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 must contain the letters 'IBM' in single quotes, which is the message prefix.
• 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.
Using the compiler user exit

- The fourth column indicates whether or not the message is to be suppressed. A '1' indicates the message is to be suppressed, and a '0' indicates that it should be printed.
- 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 (provided as one of the sample programs with the product) as a model. As you write the exit, make sure it covers the areas of initialization, message filtering, and termination.

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.

```c
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_UB_Exit_Routs, /* exit entries set at initialization */
    3 ( Uex_UB_Termination,   /* call for each msg */
      Uex_UB_Message_Filter,  /* call for each msg */
        *, *, *, * )
limited entry {
  *, /* to Uex_UB */
  *, /* to a request area */
};
```

Data Entry Fields
- **Uex_UB_Length**: Contains the length of the control block in bytes. The value is storage (Uex_UB).
- **Uex_UB_Exit_token**: Used by the user exit procedure. For example, the initialization may set it to a data structure which is used by both the message filter, and the termination procedures.
- **Uex_UB_User_char_str**: Points to an optional character string, if you specify it. For example, in pli filename (EXIT ('string'))...fn can be a character string up to thirty-one characters in length.
- **Uex_UB_User_char_len**: Contains the length of the string pointed to by the User_char_str. The compiler sets this value.
- **Uex_UB_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_UB_Filename_len**: Contains the length of the name of the source file pointed to by the Filename_str. The compiler sets this value.
Using the compiler user exit

- **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.

**Writing the initialization procedure**

Your initialization procedure should perform any initialization required by the exit, such as opening files and allocating storage. The initialization procedure-specific control block is coded as follows:

```plaintext
Dcl 1 Uex_ISA native based( null() ),
  2 Uex_ISA_Length fixed bin(31); /* storage(Uex_ISA) */
```

The global control block syntax for the initialization procedure is discussed in the section “Structure of global control blocks” on page 372.

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 is an example of a procedure-specific message filter control block:

```plaintext
Dcl 1 Uex_MFX native based( null() ),
  2 Uex_MFX_Length fixed bin(31),
  2 * 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 */
```
Using the compiler user exit

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_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. For the SQL side of the SQL preprocessor, the id is SQL. 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. The possible insert types are:
  - **Uex_Ins_Type_Xb31**: Used for an integer type and has the value 1.
  - **Uex_Ins_Type_Char**: Used for an integer type and has the value 2.
  - **Uex_Ins_Type_Series**: Used for an integer type and has the value 3.
  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
Using the compiler user exit

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.

The termination procedure-specific control block is coded as follows:

```
Dcl 1 Uex_ISA native based,
   2 Uex_ISA_Length_fixed bin(31); /* storage(Uex_ISA) */
```

The global control block syntax for the termination procedure is discussed in “Structure of global control blocks” on page 372. 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

Using the CICS run-time user exit

One of the key functions of the CICS run-time exit, CEEFXITA, is to let you control whether or not the CICS Dynamic Transaction Backout (DTB) occurs when PL/I transactions fail. The CICS run-time exit is driven immediately before and immediately after the invocation of each PL/I program within a transaction under CICS. Each time the exit is called, the typed structure CXIT (part of the include file IBMVCXT.INC) is used for communication between the PL/I run-time and the exit.

It is strongly recommended that you review and modify (if necessary) the user exit.

This structure contains information pertinent to the PL/I program, including:
- The reason for invocation (initialization or termination) of the exit.
- A reason code which indicates how the program terminated when invoked after program termination.
- Pointers to key CICS control blocks.
Using the CICS run-time user exit

**Prior to program invocation**
When the exit is invoked before the invocation of the PL/I program, the exit can tell the PL/I run-time to bypass the program invocation. In this case, DTB occurs if necessary.

During this invocation of the exit, other functions (such as interrogation of or setting of run-time options) cannot be performed.

The IBM supplied exit merely returns allowing the PL/I program invocation to proceed.

**After program termination**
When the exit is invoked after the PL/I program invocation, the exit can examine the reason for program termination and can request DTB. The termination reason code indicates why the program ended. The file IBMVCXT.INC contains detailed information.

In this case, the IBM supplied exit requests DTB if:
- The PL/I program return code (set via PLIRETC) is non-zero
- The reason for termination is anything other than normal termination

**Modifying CEEFXITA**
The following source files are supplied:

- **CEEFXITA.PLI**
  PL/I source code.
  To recompile the exit, set the INCDIR compile-time option to include the directory for IBMVCXT.INC. Enter the following command at the command line:
  ```
  pli CEEFXITA
  ```

- **IBMVCXT.INC**
  EXIT typed structure and other interface information.

- **CEEFXITA.DLL**
  Executable DLL.
  To rebuild this DLL, issue the following command from the command line:
  ```
  ilink /dll ceefxita.obj ceefxita.def
  ```

- **CEEFXITA.DEF**
  DEF file used to build CEEFXITA.DLL.

---

Using data conversion tables
The compiler, preprocessor, library, and debugger can convert both single and double bytes from ASCII to EBCDIC, and from EBCDIC to ASCII. The conversion routines are found in DLL files.

For Windows, the routines are found in these two files:
- ibmwstb.dll (non-multithreading)
- ibmwmtb.dll (multithreading)

The source for these routines, including the tables that they use, is shipped with the product so that you can use different tables if necessary. The tables are in the
product sample directory, /usr/lpp/pli/samples. If files are translated from EBCDIC to ASCII as you download them, you might want to use a different table from the one that is included.

The names of the conversion routines are IBMPBE2A (EBCDIC to ASCII), IBMPBA2E (ASCII to EBCDIC), IBMPBE4A (DBCS EBCDIC to ASCII), and IBMPBA4E (DBCS ASCII to EBCDIC). Do not change the names of the files that are included with the product.

Definition files are also supplied with the product:

For Windows, the definition files are:
• ibmwstb.def
• ibmwmtb.def

You should use these definition files when creating the corresponding DLLs.
Using data conversion tables
Chapter 19. Building dynamic link libraries

Dynamic linking is the process of resolving external references using dynamic link libraries (DLLs). Some advantages of dynamic linking are:

- Reduced memory requirements
- Simplified application modification
- Flexible software support
- Transparent migration of function
- Multiple programming language support
- Application-controlled memory usage.

DLLs are typically used to provide common functions that can be used by a number of applications. An application using a DLL can use either load-time dynamic linking or run-time dynamic linking.

You can dynamically link with the supplied run-time DLLs, as well as with your own DLLs. The following steps for creating and using a dynamic link library are described in this chapter:

- Creating the source files for a DLL
- Creating a module definition file (.DEF) for the DLL
- Compiling the source files and linking the resulting object files to build a DLL file
- Writing a module definition file to use when linking the external module that identifies what is in the DLL.

Each section contains a relevant example from the sample program SORT.PLI, which is packaged with the compiler.

Creating DLL source files

To build a DLL, you must first create source files containing the data or routines that you want to include in your DLL. No special file extension is required for DLL source files.

Each routine that you want to export from the DLL (that is, a routine that you plan to call from other executable modules or DLLs) must be an external routine, either by default or by being qualified with the external keyword.

Compiling your DLL source

You can compile your source files to create a DLL in the same way that you would compile any other file (using the PLI command) with one exception—you must compile at least one file with the DLLINIT option. You can compile every routine in a DLL with the DLLINIT option; however, no routine compiled with DLLINIT can be linked into an EXE.

You might also want to compile your programs with the option XINFO(DEF). This option creates a .DEF file for each program. These .DEF files are essential to preparing to link your DLL.
Preparing to link your DLL

When you link your DLL, you must tell the linker what names are to be exported out of the DLL.

Specifying exported names under Windows

Under Windows, you tell the linker what parts are exported using an .EXP file. The .EXP file is a binary file that is built by invoking ilib with the /GENI option and using either of the following as input:

- The .DEF file for the DLL
- All the .OBJ containing names to be exported by the DLL

Using .DEF files is preferable since it gives you control of exactly what is exported by the DLL. If you specify .OBJ names, all of the external names in the object files named are exported.

The following example shows a command you could use to create an .EXP file:

```bash
ilib /geni myliba.def
```

The Windows .DEF file created for has these characteristics:

- The Windows version contains only an EXPORTS statement
- The Windows version contains names that have been 'decorated'

The name 'decoration' depends on a routine's linkage, but if you use the .DEF files created by the compiler, you do not need to be concerned about this.

Linking your DLL

To link your DLL, use the following options and input files:

**Linker options**

- /dll,
- /out: followed by the name of your dll

**Input files**

- All of the OBJs comprising your DLL
- The .DEF or .EXP file specifying what is to be exported

For example, to link mydlla.obj and mydllb.obj into mydlla.dll, issue the following link command under Windows:

```bash
ilink /dll /out:mydlla.dll mydlla.obj mydllb.obj mydlla.exp
```

Using your DLL

Once you have built your DLL, other routines in your application can access the variables and routines exported by that DLL using one of the following methods:

- A FETCH statement
- Linking with an import library

If your application accesses an element of a DLL using a FETCH statement, you do not need to take any special action when you link. Unless your application executes that FETCH statement, the DLL does not even need to exist.

If your application accesses an element of a DLL as if it were statically linked with that DLL, then the linker must be able to resolve the name of that element.
Under Windows, the linker can resolve names from a DLL if you link with a import library for that DLL. In fact, that is how the names of PL/I library routines are resolved. For example, when you link with `ibmws20i.lib`, you are linking with the import library for `ibmws20.dll`.

Under Windows, the import library for the DLL is built when you create the .EXP file when preparing to link the DLL.

**Note:** In order for the loader to find a DLL, the DLL must reside either in your current working directory or in one of the directories listed in the PATH environment variable under Windows.

### Sample program to build a DLL

The sample programs `SORT.PLI` and `DRIVER1.PLI` show how to build and use a DLL that contains three different sorting functions. These functions keep track of the number of swap and compare operations required to do the sorting.

The files for the sample program are:

- **SORT.PLI**
  - The source file for the DLL.

- **SORT.DEF**
  - The module definition file for the DLL.

- **DRIVER1.DEF**
  - The module definition file for the executable.

- **EXTDCL.CPY**
  - The user include file.

- **DRIVER1.PLI**
  - The main program that uses SORT.DLL.

If you installed the sample programs, these files are found in the `..\SAMPLES\` directory.

Use the following sequence of commands to compile, link, and run the program:

1. `pli sort`
2. `ilib /geni sort.def`
3. `ilink /dll /out:sort.dll sort.obj sort.exp`
4. `pli driver1`
5. `ilink driver1.obj /stack:80000 sort.lib`
6. `driver1`

### Using FETCH and RELEASE in your main program

The SAMPLES directory also contains `DRIVER2.PLI` which is a modified version of `DRIVER1.PLI` that uses FETCH and RELEASE statements to dynamically link the SORT.DLL routines at run time instead of at load time.

The main advantage of using this version of the DRIVER program is that you can control when the sort routines are brought into and released from memory. Using FETCH and RELEASE statements, however, might increase your program’s execution time.
Using FETCH and RELEASE

Use the following sequence of commands to compile, link, and run this version of the DRIVER program under Windows:
1. pli sort
2. ilib /geni sort.def
3. ilink /dll /out:sort.dll sort.obj sort.exp
4. pli driver2
5. ilink driver2.obj /stack:80000
6. driver2

Exporting data from a DLL

The preceding discussion described how to export external entries from a DLL. You can also export external data from a DLL. To export external data from a DLL, the data must be declared as RESERVED throughout your application. The following conditions must also apply:

- The DLL that exports a variable must name that variable in the RESERVES option of some package in that DLL.
- All DLLs and EXEs importing a variable from another DLL must also declare that variable as RESERVED(IMPORTED).

For example, to create a DLL exporting just the variable datatab, the following routine would be used:

```pli
*process dllinit;

edata: package reserves( datatab );

dcl datatab char(256) reserved external init( .... );
end;
```

To import datatab into a procedure outside this DLL, it would be declared as:

```pli
dcl datatab char(256) reserved(IMPORTED) external;
```
Chapter 20. Using IBM Library Manager on Windows

Use the IBM Library Manager (also referred to as ILIB) to create and maintain libraries of object code, create import libraries and export object pairs, and generate module definition (.def) files. Using the ILIB utility, you can:

• Create a new library from a collection of objects
• Maintain a library
  – Add objects to an existing library
  – Delete objects from an existing library
  – Copy objects from an existing library
  – Replace objects in an existing library
• List the contents of a new or existing library
• Create import library/export object pairs from:
  – Module definition (.def) files
  – Objects generated from source files containing #pragma export and _Export statements
  – A combination of the above
• Generate module definition (.def) files from:
  – An existing DLL
  – Objects generated from source files containing #pragma export and _Export statements
  – A combination of the above

Running ILIB

Run ILIB by typing ilib at the command prompt.

You can specify parameters in the following ways:
1. Enter them directly on the command line
2. Use the ILIBenvironment variable
3. Put them in a text file, called a response file and specify the file name after the ilib command.
4. A combination of the above

You can press Ctrl+C or Ctrl+Break at any time while running ILIB to return to the operating system. Interrupting ILIB before completion restores the original library from a backup.

Notes:

1. When started, ILIB makes a backup copy of the original library in case it is interrupted or a mistake is made. Make sure you have enough disk space for both your original library and the modified copy.
2. The library must end with the extension .lib. If an extension is not specified, the default extension, .lib, will be appended. High Performance File System (HPFS) file names are supported. Hence, mylibraryname.new.lib is still a valid library.

Using the command line

You can specify all the input ILIB needs on the command line. The syntax of the command line is:

ilib [options] [libraries] [@responsefile] [objects]
Using ILIB on Windows

**Options**
Options that affect the behavior of ILIB

**Libraries**
The input library to be created or modified

**Response file**
The name of a text file containing ILIB options

**Objects**
Commands used to add, delete, replace, copy, and move object modules within the library

The ILIB command line is a free format command line; that is, the input arguments can be specified any number of times, in any order. The only exception is the /FREEFORMAT option, which does have a position restriction. See “/FREEFORMAT” on page 392 for more information.

**Note:** For compatibility with the OS/2 release of ILIB, a fixed format command line is also supported. To use the fixed format command line, the /NOFREEFORMAT option must be specified immediately following ilib on the command line, or as the first parameter in the ILIB environment variable. The default command line format is free format.

For the purposes of this document, only the free format command line will be described in detail.

**Using the ILIB environment variable**

You can use the ILIB environment variable to specify any default ILIB options. When the ilib command is invoked, the environment variable will be parsed before the command line.

Use the SET command to give value to the ILIB environment variable. You can do this in the following ways:

**Command line**
When the SET command is used on the command line, the values you specify are in effect for only that session. They override values previously specified.

You can append the original value of the variable using %variable%. The following example would cause the ILIB environment variable to be set to the original value of the ILIB environment variable, with the /NOFREEFORMAT option specified ahead of any existing options.

```
SET ILIB=/FREEFORMAT %ILIB%
```

**Windows control panel**

Windows allows you to update environment variables and have them take effect immediately (that is, no reboot required) using the Windows Control Panel.

To set the ILIB environment variable:
- Select the **Main** group by double-clicking on the **Main** icon.
- Select the **System** icon from the **Main** group by double-clicking on it.
- Enter ILIB in the **Variable** field.
- Enter the value for the ILIB environment variable in the **Value** field.
- Choose **Set**.
Windows 98 AUTOEXEC.BAT file

Windows 98 allows you to set environment variables in the AUTOEXEC.BAT file. Any environment variables set in this fashion are available in every user session.

Add a line to your AUTOEXEC.BAT file that sets the environment variable to the value you want. Consider the following example:

```
SET ILIB=/NOBACKUP
```

Because environment variables specified in your AUTOEXEC.BAT file are in effect for every session you start, this is a good place to specify options that you want to apply each time you invoke ILIB. However, after you make a change to your AUTOEXEC.BAT file, you must reboot your system to have the change take effect.

Using an ILIB response file

To provide input to ILIB with a response file, type:

```
ilib @responsefile
```

The `responsefile` is the name of a file containing the same information that can be specified on the command line.

Why use a response file?

Use a response file for:
- Complex and long commands you type frequently
- Strings of commands that exceed the limit for command line length.

A response file extends the command line to include everything in the response file. To split input to ILIB between the command line and a response file, put part of your input on the command line and specify a response file (preceding the response file name with the at sign (`@`)). No space can appear between the at sign and the file name.

The response file name can be any valid Windows file name. To use special characters in the file name, such as a space or the `@` symbol, the file name must be enclosed in quotes.

ILIB responds to input you place in a response file just as it does to input you enter on a command line. Any newline characters that occur between arguments are treated as spaces. This allows you to extend an ILIB command to multiple lines.

Note: The options which specify which format command line to use (`/FREEFORMAT` or `/NOFREEFORMAT`) must be specified as the first parameter following `ilib` on the command line or as the first parameter in the ILIB environment variable. They cannot be specified inside the response file.

Examples specifying ILIB parameters

The following examples show different methods for specifying parameters to ILIB.

The operations shown in each example create a new library, `newlib.lib`, and its listing file, `newlib.lst`, from the existing `mylib.lib` library. `mylib.lib` is unchanged, but `newlib.lib` has these changes:
- The module text is deleted
Using ILIB on Windows

- The object file root.obj is appended as an object module with the name root
- The module table is deleted and is replaced by a new table which is appended after root
- The module string is copied into an object file named string.obj

Command Line Method

At the command line prompt, enter the following:

```
ilib /out:newlib.lib /list:newlib.lst mylib.lib /remove:text root table
/extract:string
```

Response File Method

First, create a response file with the following contents:

```
/out:newlib.lib
/list:newlib.lst
mylib.lib
/remove:root
table
/extract:string
```

Then, assuming the name of the response file is `response.fil`, invoke ILIB with:

```
ilib @response.fil
```

Controlling ILIB input

ILIB determines the format of any input files by examining the file contents. Most file formats can be identified by the file header information. If the format of an input file is not recognized and seems to contain only ASCII, it is assumed to be a module definition (.def) file.

ILIB allows you to place any extension you choose on a file and still have it dealt with correctly.

Controlling ILIB output

ILIB determines what output is to be produced by examining the options that you supply on the command line. The following options control ILIB output:

**Option Description**

```
/O[UT]:filename
   A static library is produced.

/GEND[E]F]:filename
   A module definition (.def) file is produced. The short form, /gd, may also be used.

/GENI[MPLIB]:filename
   An import library/export object pair is produced. The short form, /gi, may also be used.

/L[IST]:filename
   A list file is produced.
```

If none of the above are specified, ILIB will determine what is to be produced, as follows:

- If a DEF file is input to ILIB, an import library/export object pair will be produced.
**Using ILIB on Windows**

**Note:** If there are no exported symbols, then no import library will be produced.

- If a library and/or object(s) are input to ILIB, a library combining them will be produced.

ILIB will allow you to generate a DEF file directly from a DLL. However, since the only information that a DLL has in it is the undecorated (exported) names, symbol decoration (calling convention) and type information (function or data) cannot be determined. ILIB will assume that all symbols exported from the DLL are _Optlink (the default linkage convention), unless an object file is provided that indicates otherwise.

The best way of using ILIB with a DLL is to use ILIB to create a DEF file using the /gd option. Edit the DEF file to change decorations, where appropriate, and then run the DEF file through ILIB using the /gi option to produce an import library/export object pair.

If an import library/export object pair is requested, and only a DLL is specified as input, ILIB will generate an error.

**Controlling ILIB output**

The following are examples showing how to control ILIB output.

**Library**

The following example creates create the library newlib.lib out of the objects in text.obj and mylib.lib.

```
ilib /out:newlib.lib text.obj mylib.lib
```

**Note:** Unless newlib.lib is specified as an input file, its contents will not be included in the library. If an output file already exists, and is not used as an input file, it will be replaced.

**DEF File**

This example creates the module definition file winner.def from the DLL winner.dll.

```
ilib /gd:winner.def winner.dll
```

**Import Library/Export Oject Pair**

The following example creates an import library named winner.lib and an export object named winner.exp. However, if no exported symbols are contained in winner.def, then winner.lib will not be produced.

```
ilib /gi winner.def
```

**List File**

The following example generates the list will generate the list file mylib.lst, based on the library mylib.lib, in the current directory.

```
ilib /list:mylib.lst mylib.lib
```
Using ILIB on Windows

ILIB objects

ILIB objects are used to manipulate modules in a library. When you run ILIB, you can specify multiple objects in any order.

Each object consists of the ILIB command, followed by the name of the object module that is the subject of the command. Separate objects on the command line with a space or tab character.

Summary of ILIB objects

The following is a summary of ILIB objects on Windows.

Table 27. ILIB objects on Windows

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Add/replace the named object in the library</td>
<td>None</td>
<td>&quot;Add/Replace&quot;</td>
</tr>
<tr>
<td>/E[XTRACT]:obj</td>
<td>Copy the named object into the current directory and overwrite it if it already exists</td>
<td>None</td>
<td>&quot;/EXTRACT&quot; on page 389</td>
</tr>
<tr>
<td>/R[EMOVE]:obj</td>
<td>Remove the named object from the list of objects to be placed in the output library</td>
<td>None</td>
<td>&quot;/REMOVE&quot; on page 390</td>
</tr>
</tbody>
</table>

Notes:

1. ILIB objects are not case sensitive, so you can specify them in lower-, upper-, or mixed-case.
   You can also substitute a dash (-) for the slash (/) preceding the object. For example, -REMOVED:filename is equivalent to /REMOVE:filename.
2. You can specify objects in either short or long form. For example, /R:filename and /RE:filename are equivalent to /REMOVE:filename.
3. The order of operations when processing the command line is left to right.
4. ILIB never makes changes to your input library while it runs. It copies the library and makes changes to the copy. If ILIB is interrupted, your original library will be restored.
   If you do not specify an output library, ILIB will not produce any output.

Add/Replace

The default action, when filename is specified on the command line without an associated object, is to add it to the library. If filename already exists in the library, it will be replaced.

Adding an Object Module to a Library

Type the name of the object file to be added on the command line. The .obj extension may be omitted.
ILIB uses the base name of the object file as the name of the object module in the library. For example, if the object file cursor.obj is added to a library file, the name of the corresponding object module is cursor.

Object modules are always added to the end of a library file.

Replacing an Object Module in a Library

Type the name of the object module to be replaced on the command line. The .obj extension may be omitted.

If the object module already exists in the library, ILIB will replace it with the new copy.

Combining Two Libraries

Specify the name of the library file to be added, including the .lib extension, on the command line. A copy of the contents of that library is added to the library file being modified. If both libraries contain a module with the same name, ILIB generates a warning message, and uses only the first module with that name.

ILIB adds the modules of the library to the end of the library being changed. The added library still exists as an independent library because ILIB copies the modules without deleting them.

Examples

The following command adds the file sample.obj to the library mylib.lib. If sample.obj already exists in the library mylib.lib, ILIB will replace it.

```
ilib /out:mylib.lib mylib.lib sample.obj
```

This example adds the contents of the library mylib.lib to the library newlib.lib. The library mylib.lib is unchanged after this command is executed.

```
ilib /out:newlib.lib newlib.lib mylib.lib
```

Use /EXTRACT to copy a module from the library into an object file of the same name. The module remains in the library.

When ILIB copies the module to an object file, it adds the .obj extension to the module name and places the file in the current directory. If a file with this name already exists, ILIB overwrites it.

Example The command above copies the module sample from the mylib.lib library to a file called sample.obj in the current directory. The module sample in mylib.lib is not altered.

```
ilib mylib.lib /extract:sample
```
Using ILIB on Windows

**/REMOVE**

```
>>> /REMOVE:obj
```

Use /REMOVE to delete an object module from a library. After /REMOVE, specify the name of the module to be deleted. Module names do not have path names or extensions.

**Example**

The following command deletes the module sample from the library mylib.lib.

```
ilib /out:mylib.lib mylib.lib /remove:sample
```

This next command copies sample.obj from the mylib.lib library to an object file in the current directory. Then sample.obj is deleted from the library.

```
ilib /out:mylib.lib mylib.lib /extract:sample /remove:sample
```

**ILIB options**

ILIB options affect the behavior of ILIB. When you run ILIB, you can specify multiple options in any order. The only exception is the /FREEFORMAT option, which has a position restriction.

Separate options on the command line with a space or tab character.

**Summary of ILIB options**

The following is a summary of ILIB options on Windows.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>/?</td>
<td>Display help</td>
<td>None</td>
<td>&quot;/?&quot; on page 391</td>
</tr>
<tr>
<td>/BA[CKUP]</td>
<td>Back up the output file (if it exists) before overwriting it</td>
<td>/BA</td>
<td>&quot;/BACKUP&quot; on page 391</td>
</tr>
<tr>
<td>/NOBA[CKUP]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/DEF:def</td>
<td>Specify the name of a .def file to use to get information about exported symbols and linker parameters</td>
<td>None</td>
<td>&quot;/DEF&quot; on page 391</td>
</tr>
<tr>
<td>/F[REEFORMAT]</td>
<td>Use the free format command line</td>
<td>/F</td>
<td>&quot;/FREEFORMAT&quot; on page 392</td>
</tr>
<tr>
<td>/NOF[REEFORMAT]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/GEN[DEF]:filename</td>
<td>Generate a .def file</td>
<td>None</td>
<td>&quot;/GENDEF&quot; on page 392</td>
</tr>
<tr>
<td>/GEN[I[MPLIB]:filename</td>
<td>Generate an import library</td>
<td>None</td>
<td>&quot;/GI&quot; on page 392</td>
</tr>
<tr>
<td>/H[ELP]</td>
<td>Display help</td>
<td>None</td>
<td>&quot;/HELP&quot; on page 392</td>
</tr>
<tr>
<td>/L[IST]:filename</td>
<td>Generate a list file</td>
<td>None</td>
<td>&quot;/LIST&quot; on page 393</td>
</tr>
</tbody>
</table>
### Using ILIB on Windows

Table 28. ILIB options on Windows (continued)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Default</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>/NOE[XTDICTIONARY]</td>
<td>Do not generate an extended dictionary in an OMF library</td>
<td>/EXTD</td>
<td>393</td>
</tr>
<tr>
<td>/EXTD[ICTIONARY]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/O[UT]:filename</td>
<td>Specify the name of the output library</td>
<td>None</td>
<td>393</td>
</tr>
<tr>
<td>/QUIET, /NLOGO</td>
<td>Do not display the banner on startup</td>
<td>/LO</td>
<td>393</td>
</tr>
<tr>
<td>/LOGO, /NOQUIET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/WARN:msgnum, msgnum[...]]</td>
<td>Enable printing of warning message number</td>
<td>None</td>
<td>393</td>
</tr>
<tr>
<td>/NOWARN:msgnum, msgnum[...]]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. ILIB options are not case sensitive, so you can specify them in lower-, upper-, or mixed-case.
   You can also substitute a dash (-) for the slash (/) preceding the option. For example, -FREEFORMAT is equivalent to /FREEFORMAT.

2. You can specify options in either short or long form. For example, /F, /FR, and /FREE are equivalent to /FREEFORMAT.

See below for detailed information on each ILIB option.

/?

Use /? to display a list of valid ILIB options. This option is equivalent to /HELP.

/BACKUP

Use /BACKUP to back up the output file (if it exists) before overwriting it.

ILIB uses the base name of the library as the name of the backup library, and then appends the .bak extension. For example, if the library being modified is mylib.lib and a backup is requested, ILIB will create mylib.bak in the current directory.

/DEF

Use /DEF to specify the name of the .def file to use to get information about exported symbols and linker parameters.
Using ILIB on Windows

This option is not required, since ILIB will recognize .def files by their contents if they are placed with other input files on the command line.

/FREEFORMAT

Use the /FREEFORMAT option to tell ILIB that you are using the free format command line. The free format command line allows you to specify ILIB input arguments any number of times, in any order.

Note: This option must be specified immediately following lib on the command line, or as the first argument in the ILIB environment variable. If you don't specify either /FREEFORMAT or /NOFREEFORMAT, ILIB will default to the free format command line.

/GENDEF

Use the /GENDEF option to create a module definition (.def) file.

Example

The following command creates the module definition file sample.def from the DLL sample.dll.

    lib /gd:sample.def sample.dll

/GI

Use the /GENIMPLIB option to create an import library/export object pair.

Example

The command above will create an import library named sample.lib and an export object named sample.exp from the module definition file sample.def. However, if no exported symbols are contained, then sample.lib will not be produced.

    lib /gi sample.def

/HELP

Use /HELP to display a list of valid ILIB options. This option is equivalent to /?.
/LIST

Use the /LIST option to generate a list file. If filename is not specified, ILIB will add the extension .lst to the input filename.

Example

The following command directs ILIB to place a listing of the contents of mylib.lib into the file mylib.lst. No path specification is given for mylib.lst. By default, the file created is put into the current directory.

    ilib mylib /list:mylib.lst

Note: The /LISTLEVEL option is not supported in the Windows release of ILIB.

/NOEXT

Use /NOEXTDICTIONARY to disable generation of the extended dictionary.

The extended dictionary is an optional part of the library that increases linking speed. However, using an extended dictionary requires more memory. The space reserved for the extended dictionary is limited to 64K. If ILIB reports an out-of-memory error, you may want to use this option. As an alternative, you can split large libraries into smaller libraries to use in linking.

/OUT

Use the /QUIET or /NOLOGO options to suppress the ILIB copyright notice.

/WARN

Use the /WARN option to enable printing of the message number specified in the msgnum parameter.
Chapter 21. Calling conventions

This chapter discusses the calling conventions used by PL/I for Windows:

- OPTLINK
- SYSTEM
- STDCALL
- CDECL

The OPTLINK linkage convention (see “OPTLINK linkage” on page 396 for details) is also supported by VisualAge for C++ (OS/2 and Windows) and is the fastest method of calling PL/I procedures, C functions, or assembler routines. OPTLINK is not, however, standard for all Windows applications.

On Windows, specifying SYSTEM linkage is synonymous with STDCALL linkage and is implemented the same as STDCALL. The compiler, however, considers the names SYSTEM and STDCALL to be distinct and complains if you mix them. The STDCALL linking convention is described in “STDCALL linkage (Windows only)” on page 405.

You can specify the calling convention for all functions within a program using the LINKAGE suboption of the DEFAULT compile-time option. You can also use the LINKAGE option of the OPTIONS attribute to specify the linkage for individual functions.

Note: You cannot call a function using a different calling convention than the one with which it is compiled. For example, if a function is compiled with SYSTEM linkage, you cannot later call it specifying OPTLINK linkage.

Understanding linkage considerations

On Windows, there are three primary linkages that the PL/I compiler supports: OPTLINK, CDECL, and STDCALL. On Windows, all the system services use the STDCALL linkage.

These linkages differ in their parameter passing conventions:
- The OPTLINK linkage is the only one that attempts to pass some parameters in registers; the other linkages pass all the parameters on the stack.
- The STDCALL linkage is the only one that makes the callee responsible for cleaning up the stack; the other linkages make the caller responsible.

The PL/I for Windows compiler interprets any specification of the SYSTEM linkage as if the STDCALL linkage were intended. The VisualAge C compiler does the same.

On Windows, all external names are decorated. If the external attribute does not specify a name, the name decoration depends on the linkage:
- Routines with the CDECL linkage have a ‘_’ added as a prefix so that, for example, the name FUNKY would become _FUNKY.
- Routines with the OPTLINK linkage have a ‘?’ added as a prefix so that, for example, the name FUNKY would become ?FUNKY.
- Routines with the STDCALL linkage have a ‘_’ added as a prefix and a ‘@’ followed by the bytes used by its parameters added as a suffix. For example,
Understanding linkage considerations

then, if the name FUNKY had two byvalue pointers or any two byaddr parameters, it would become _FUNKY@8.

One consequence of these name decorations is that if a caller of a routine specifies the wrong linkage for that routine, the program fails to link.

So far, the discussion of name decoration has applied only to routines for which the external attribute did not specify a name. It also applies when the external attribute specifies a name that differs only in case from the declared name. In these situations, the name specified as part of the external attribute is decorated.

For example, given the following declare, the name that the linker sees is ?getenv.

dcl getenv ext('getenv')
   entry( char(*) varz byaddr nonasgn )
   returns( pointer )
   options( nodescriptor linkage(optlink) );

Similarly, for the following declare (for the Windows system routine that loads a DLL), the name specified as part of the external attribute is decorated, and the linker sees the name as _LoadLibraryA@4.

dcl loadlibrarya ext('LoadLibraryA')
   entry( char(*) varz byaddr nonasgn )
   returns( pointer byvalue )
   options( linkage(stdcall) nodescriptor );

If, however, a name is specified as part of the external attribute and that name differs from the declared name by more than its case, then no name decoration occurs.

For example, given the following declare, no name decoration occurs and the name that the linker sees is ?getenv.

dcl getenv ext('?getenv')
   entry( char(*) varz byaddr nonasgn )
   returns( pointer )
   options( nodescriptor linkage(optlink) );

Performing name decoration yourself as illustrated in this last example usually makes your code less portable. For instance, only the first declare for getenv in the preceding examples is valid for Windows and AIX.

OPTLINK linkage

This is the default calling convention. It is an alternative to SYSTEM linkage that is normally used for calls to the operating system. This linkage provides better total performance than SYSTEM linkage.

Features of OPTLINK

The OPTLINK convention has the following features:

- Parameters are pushed from right to left onto the stack.
- The caller cleans up the stack.
- The general-purpose registers EBX, EDI, and ESI are preserved across the call.
- The general-purpose registers EAX, ECX, and EDX are not preserved across the call.
- Floating-point registers are not preserved across the call.
The three conforming parameters that are lexically leftmost (conforming parameters are the addresses for all BYADDR parameters and the following BYVALUE parameters: pointer, handle, ordinal, offset, limited entry, real fixed binary, character(1), and nonvarying bits occupying 1 byte or less) are passed in the three unpreserved general-purpose registers.

Up to four real floating-point or two complex parameters (the lexically first four) are passed in extended precision format (80-bit) in the floating-point register stack.

All conforming parameters not passed in registers and all nonconforming parameters are passed on the 80386 stack.

Space for the parameters in registers is allocated on the stack, but the parameters are not copied into that space.

Conforming return values are returned in EAX.

Real floating-point return values are returned in extended precision format in the topmost register of the floating-point stack.

Complex floating-point return values are returned in extended precision format in the topmost two registers of the floating-point stack.

When you are calling external functions, the floating-point register stack contains only valid parameter registers on entry, and valid return values on exit.

Functions returning aggregates pass the address of a storage area determined by the caller as a hidden parameter. This area becomes the returned aggregate. The address of this aggregate is returned in EAX.

The direction flag must be clear upon entry to functions, and clear on exit from functions. The state of the other flags is ignored on entry to a function, and undefined on exit.

The compiler does not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

**Tips for using OPTLINK**

By following the tips given below when you use OPTLINK linkage, you can improve the performance of your applications.

- The conforming and floating-point parameters that are most heavily used should be lexically leftmost in the parameter list so they will be considered for registers first. If they are adjacent to each other, the preparation of the parameter list will be faster.

- If you have a parameter that is used near the end of a function, put it at or near the end of the parameter list. If all of your parameters are used near the end of functions, consider using SYSTEM linkage.

- Compile with OPTIMIZE. (See “OPTIMIZE” on page 99.)

**General-purpose register implications**

**Parameters**

EAX, EDX, and ECX are used for the lexically first three conforming parameters with EAX containing the first parameter, EDX the second, and ECX the third. Four bytes of stack storage are allocated for each register parameter that is present, but the parameters exist only in the registers at the time of the call.
Examples of passing parameters

The following examples are included only for purposes of illustration and clarity and have not been optimized. These examples assume that you are familiar with programming in assembler. In each example, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

Passing conforming parameters to a routine

The following example shows the code sequences and a picture of the stack for a call to the function FUNC1. It is assumed that this program is compiled with the PREFIX(NOFIXEDOVERFLOW) option.

dcl func1 entry( char(1),
    fixed bin(15),
    fixed bin(31),
    fixed bin(31) )
returns( fixed bin(31) )
options( byvalue nodescriptor );

dcl x fixed bin(15);
dcl y fixed bin(31);
y = func1('A', x, y+x, y);

caller's Code Up Until Call:
PUSH y ; Push p4 onto the 80386 stack
SUB ESP, 12 ; Allocate stack space for
; register parameters
MOV AL, 'A' ; Put p1 into AL
MOV DX, x ; Put p2 into DX
MOVSX ECX, DX ; Sign-extend x to long
ADD ECX, y ; Calculate p3 and put it into ECX
CALL FUNC1 ; Make call

callee's Prolog Code:
PUSH EBP ; Save caller's EBP
MOV EBP, ESP ; Set up callee's EBP
SUB ESP, callee's local size ; Allocate callee's Local
PUSH EBX ; Save preserved registers -
PUSH EDI ; will optimize to save
PUSH ESI ; only registers callee uses

Stack Just After Call  Register Set Just After Call

<table>
<thead>
<tr>
<th>caller's Local</th>
<th>EAX</th>
<th>undefined</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blank Slot For p3</td>
<td>ECX</td>
<td>p3</td>
<td></td>
</tr>
<tr>
<td>Blank Slot For p2</td>
<td>EDX</td>
<td>undefined</td>
<td>p2</td>
</tr>
<tr>
<td>Blank Slot For p1</td>
<td>EDI</td>
<td>caller's EDI</td>
<td></td>
</tr>
<tr>
<td>caller's EIP</td>
<td>ESI</td>
<td>caller's ESI</td>
<td></td>
</tr>
</tbody>
</table>

ESP ───/SM590000 └───────────────────┘ └─────────┴────────┘

General-purpose register implications
The term "undefined" in registers EBX, EDI and ESI refers to the fact that they can be safely overwritten by the code in FUNCTION1.

Callee's Epilog Code:

```
MOV EAX, RetVal ; Put return value in EAX
POP ESI ; Restore preserved registers
POP EDI
POP EBX
MOV ESP, EBP ; Deallocate callee's local
POP EBP ; Restore caller's EBP
RET ; Return to caller
```
General-purpose register implications

Passing floating-point parameters to a routine:

The following example shows code sequences, 80386 stack layouts, and floating-point register stack states for a call to the routine FUNC2. For simplicity, the general-purpose registers are not shown. It is assumed that this program is compiled with the IMPRECISE option.

dcl func2 entry( float bin(21),
    float bin(53),
    float bin(64),
    float bin(21),
    float bin(53) )
returns( float bin(53) )
options( byvalue nodescriptor );

dcl (a, b, c) float bin(53);
dcl (d, e) float bin(21);

\[ a = b + \text{func2}(a, d, \text{prec}(a + c, 53), e, c); \]

caller's Code Up Until Call:

```assembly
PUSH 2ND DWORD OF c ; Push upper 4 bytes of c onto stack
PUSH 1ST DWORD OF c ; Push lower 4 bytes of c onto stack
FLD DWORD_PTR e ; Load e into 80387, promotion
    ; requires no conversion code
FLD QWORD_PTR a ; Load a to calculate p3
FADD ST(0), QWORD_PTR c ; Calculate p3, result is float bin(64)
    ; from nature of 80387 hardware
FLD QWORD_PTR d ; Load d, no conversion necessary
FLD QWORD_PTR a ; Load a, demotion requires conversion
FSTP DWORD_PTR [EBP - TI] ; Store to a temp (TI) to convert to float
FLD DWORD_PTR [EBP - TI] ; Load converted value from temp (TI)
SUB ESP, 32 ; Allocate the stack space for
    ; parameter list
CALL FUNC2 ; Make call
```
General-purpose register implications

Stack Just After Call  
<table>
<thead>
<tr>
<th>caller's Local</th>
<th>80387 Register Set Just After Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper Dword of p5</td>
<td>ST(7) Empty</td>
</tr>
<tr>
<td>Lower Dword of p5</td>
<td>ST(6) Empty</td>
</tr>
<tr>
<td>Blank Dword for p4</td>
<td>ST(5) Empty</td>
</tr>
<tr>
<td>Four</td>
<td>ST(4) Empty</td>
</tr>
<tr>
<td>Blank</td>
<td>ST(3) Empty p4 (e)</td>
</tr>
<tr>
<td>Dwords</td>
<td>ST(2) Empty p3 (a + c)</td>
</tr>
<tr>
<td>for p3</td>
<td>ST(1) Empty p2 (d)</td>
</tr>
<tr>
<td>Two Blank</td>
<td>ST(0) Empty p1 (a)</td>
</tr>
<tr>
<td>Dwords for p2</td>
<td></td>
</tr>
<tr>
<td>Blank Dword for p1</td>
<td></td>
</tr>
<tr>
<td>caller's EIP</td>
<td></td>
</tr>
</tbody>
</table>

ESP ───/SM590000 └────────────────────┘

callee's Prolog Code:

```plaintext
PUSH EBP ; Save caller's EBP
MOV EBP, ESP ; Set up callee's EBP
SUB ESP, callee's local size ; Allocate callee's Local
PUSH EBX ; Save preserved registers -
PUSH EDI ; will optimize to save
PUSH ESI ; only registers callee uses
```
General-purpose register implications

Stack After Prolog

80387 Register Set After Prolog

| caller's Local | ST(7) | Empty |
| Upper Dword of p5 | ST(6) | Empty |
| Lower Dword of p5 | ST(5) | Empty |
| Blank Dword for p4 | ST(4) | Empty |
| Four | ST(3) | p4 |
| Blank | ST(2) | p3 |
| Dwords for p3 | ST(1) | p2 |
| Two Blank Dwords for p2 | ST(0) | p1 |

ESP ───/SM590000 └───────────────────┘

Callee's Epilog Code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD RETVAL</td>
<td>Load return value onto floating-point stack</td>
</tr>
<tr>
<td>POP ESI</td>
<td>Restore preserved registers</td>
</tr>
<tr>
<td>POP EDI</td>
<td></td>
</tr>
<tr>
<td>POP EBX</td>
<td></td>
</tr>
<tr>
<td>MOV ESP, EBP</td>
<td>Deallocate callee's local</td>
</tr>
<tr>
<td>POP EBP</td>
<td>Restore caller's EBP</td>
</tr>
<tr>
<td>RET</td>
<td>Return to caller</td>
</tr>
</tbody>
</table>
General-purpose register implications

<table>
<thead>
<tr>
<th>Stack After Epilog</th>
<th>80387 Register Set After Epilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller's Local</td>
<td>ST(7) Empty</td>
</tr>
<tr>
<td>Upper Dword of p5</td>
<td>ST(6) Empty</td>
</tr>
<tr>
<td>Lower Dword of p5</td>
<td>ST(5) Empty</td>
</tr>
<tr>
<td>Blank Dword for p4</td>
<td>ST(4) Empty</td>
</tr>
<tr>
<td>Four</td>
<td>ST(3) Empty</td>
</tr>
<tr>
<td>Blank</td>
<td>ST(2) Empty</td>
</tr>
<tr>
<td>Dwords</td>
<td>ST(1) Empty</td>
</tr>
<tr>
<td>for p3</td>
<td>ST(0) Return Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack After Cleanup</th>
<th>80387 Register Set After Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller's Local</td>
<td>ST(7) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(6) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(5) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(4) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(3) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(2) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(1) Empty</td>
</tr>
<tr>
<td></td>
<td>ST(0) Return Value</td>
</tr>
</tbody>
</table>

**SYSTEM linkage**

To use this linkage convention, you must specify the OPTIONS(LINKAGE(SYSTEM)) attribute in the declaration of the function, or specify the DEFAULT(LINKAGE(SYSTEM)) compile-time option.

**Features of SYSTEM**

The following rules apply to the SYSTEM linkage convention:

- All parameters are passed on the 80386 stack.
- Parameters are pushed onto the stack in right-to-left order.
SYSTEM linkage

- The calling function is responsible for removing parameters from the stack.
- All parameters are doubleword (4-byte) aligned.
- Values are returned in the same manner as the OPTLINK linkage.
- The direction flag must be clear upon entry to functions and clear on exit from functions. The state of the other flags is ignored on entry to a function, and undefined on exit.
- The compiler does not change the contents of the floating-point control register.
  If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

Example using SYSTEM linkage

The following example is included only for purposes of illustration and clarity and has not been optimized. The example assumes that you are familiar with programming in assembler. In the example, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

The following example shows the code sequences and a picture of the stack for a call to the function FUNC3 which has two local variables, x and y (both fixed bin(31)). For the call
dcl func3 entry( fixed bin(31),
  fixed bin(31),
  fixed bin(31) )
returns( fixed bin(31) )
options( byvalue nodescriptor linkage(system) );
m = func3(a,b,c);
the stack for the call to FUNC3 would look like this:

```
Stack
  c
  b
  a
caller's EIP
caller's EBP
  x
  y
Saved EDI
Saved ESI
Saved EBX
```

Higher Memory

Lower Memory

These would only be pushed if they were used in this function.

The instructions used to build this activation record on the stack look like this on the calling side:
PUSH c
PUSH b
PUSH a
For the callee, the code looks like this:

```assembly
func3 PROC
    PUSH EBP
    MOV EBP, ESP ; Allocating 8 bytes of storage
    SUB ESP, 8 ; for two local variables.
    PUSH EDI ; These would only be
    PUSH ESI ; pushed if they were used
    PUSH EBX ; in this function.
    .
    MOV EAX, [EBP - 8] ; Load y into EAX
    MOV EBX, [EBP + 12] ; Load b into EBX
    .
    XOR EAX, EAX ; Zero the return value
    POP EBX ; Restore the saved registers
    POP ESI
    POP EDI
    LEAVE ; Equivalent to MOV ESP, EBP
    ; POP EBP
    RET
func3 ENDP
```

The saved register set is EBX, ESI, and EDI. The other registers (EAX, ECX, and EDX) can have their contents changed by a called routine.

Under some circumstances, the compiler does not use EBP to access automatic and parameter values, thus increasing the application’s efficiency. Whether it is used or not, EBP does not change across the call.

When passing aggregates by value, the compiler generates code to copy the aggregate on to the 80386 stack. If the size of the aggregate is larger than an 80386 page size (4K), the compiler generates code to copy the aggregate backward (that is, the last byte in the aggregate is the first to be copied).

Aggregates are not returned on the stack. The caller pushes the address where the returned aggregate is to be placed as a lexically first hidden parameter. A function that returns an aggregate must be aware that all parameters are 4 bytes farther away from EBP than they would be if no aggregate return were involved. The address of the returned aggregate is returned in EAX.

**STDCALL linkage (Windows only)**

To use this linkage convention, you must specify the OPTIONS(LINKAGE(STDCALL)) attribute in the declaration of the function, or specify the DEFAULT(LINKAGE(STDCALL)) compile-time option.
Features of STDCALL

The following rules apply to the STDCALL calling convention:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The called function removes the parameters from the stack.
- Floating point values are returned in ST(0), the top register of the floating point register stack. Functions returning aggregate values return them as follows:

  Size of Aggregate  Value Returned in
  8 bytes        EAX-EDX pair
  5, 6, 7 bytes   EAX The address to place the return values is passed as a hidden parameter in EAX.
  4 bytes        EAX
  3 bytes        EAX The address to place the return values is passed as a hidden parameter to EAX.
  2 bytes        AX
  1 byte         AL

For functions that return aggregates 5, 6, 7 or more than 8 bytes in size, the address to place the return values is passed as a hidden parameter, and the address is passed back in EAX.

- STDCALL has the restriction that an unprototyped STDCALL function with a variable number of arguments will not work.
- Function names are decorated with an underscore prefix, and a suffix which consists of an at sign (@), followed by the number of bytes of parameters (in decimal). Parameters of less than four bytes are rounded up to four bytes. Structure sizes are also rounded up to a multiple of four bytes. For example, consider a function fred prototyped as follows:

```pli
  dcl fred ext entry (fixed bin(31) byvalue, fixed bin(31) byvalue,
                   fixed bin(15) byvalue);
```

It would appear as follows in the object module:

```plaintext
_FRED@12
```

When building export lists in .DEF files, the decorated version of the name should be used. If you use undecorated names in the DEF file, you must give the object files to ILIB along with the DEF file. ILIB uses the object files to determine how each name ended up after decoration.

Examples using the STDCALL convention

The following examples are included for purposes of illustration and clarity only. The examples assume that you are familiar with programming in assembler. In the examples, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.
For the following call, \(a, b,\) and \(c\) are 32-bit integers and \(\text{func}\) has two local variables, \(x\) and \(y\) (both 32-bit integers):

\[ m = \text{func}(a,b,c) \]

The stack for the call to \(\text{FUNC}\) would look like this:

\[
\begin{array}{c}
\text{EBP} \\
\text{ESP} \\
\text{c} \\
\text{b} \\
\text{a} \\
\text{caller's EIP} \\
\text{caller's EBP} \\
\text{x} \\
\text{y} \\
\text{Saved EDI} \\
\text{Saved ESI} \\
\text{Saved EBX} \\
\end{array}
\]

These would only be pushed if they were used in this function.

The instructions used to create this activation record on the stack look like this on the calling side:

\[
\begin{align*}
PUSH & \ c \\
PUSH & \ b \\
PUSH & \ a \\
\text{CALL} & \ _\text{func}@12 \\
\ldots & \\
\text{MOV} & \ m, \text{EAX} \\
\ldots & \\
\end{align*}
\]

For the callee, the code looks like this:

\[
\begin{align*}
&\_\text{func}@12 \ \text{PROC} \\
&PUSH \ EBP \\
&MOV \ EBP, \ ESP ; \text{Allocating 8 bytes of storage} \\
&SUB \ ESP, 8 ; \text{for two local variables.} \\
PUSH & \ ESI ; \text{These would only be} \\
PUSH & \ EDI ; \text{pushed if they were used} \\
PUSH & \ EBX ; \text{in this function.} \\
\ldots & \\
&MOV \ EAX, [EBP - 8] ; \text{Load } y \text{ into EAX} \\
&MOV \ EBX, [EBP + 12] ; \text{Load } b \text{ into EBX} \\
\ldots & \\
&XOR \ EAX, \ EAX ; \text{Zero the return value} \\
&POP \ EBX ; \text{Restore the saved registers} \\
&POP \ ESI \\
&POP \ EDI \\
&\text{LEAVE} ; \text{Equivalent to} \\
&\text{RET} \ 0CH \\
&\_\text{func}@12 \ \text{ENDP}
\end{align*}
\]
STDCALL linkage

The saved register set is EBX, ESI, and EDI.

Structures are not returned on the stack. The caller pushes the address where the returned structure is to be placed as a lexically first hidden parameter. A function that returns a structure must be aware that all parameters are four bytes farther away from EBP than they would be if no structure were involved. The address of the returned structure is returned in EAX.

Using WinMain (Windows only)

You can use WinMain by specifying OPTIONS(WINMAIN) on the procedure statement (see the PL/I Language Reference for syntax). This automatically implies LINKAGE(STDCALL) and EXT('WinMain').

Your WinMain routine needs four parameters:
- An instance handle
- A previous handle
- A pointer to the command line
- An integer to be passed to ShowWindow

These are the same four parameters expected by WinMain in C. The calls made inside this routine are the same as those expected from a C routine.

An example guisamp.pli is provided in the samples directory (see the program prolog for more details)

CDECL linkage

To use this linkage convention, you must specify the OPTIONS(LINKAGE(CDECL)) attribute in the declaration of the function, or specify the DEFAULT(LINKAGE(CDECL)) compile-time option.

Features of CDECL

The following rules apply to the CDECL calling convention:
- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The calling function removes the parameters from the stack.
- Floating point values are returned in ST(0). All functions returning non-floating point values return them in EAX, except for the special case of returning aggregates less than or equal to eight bytes in size. For functions that return aggregates less than or equal to four bytes in size, the values are returned as follows:
  
<table>
<thead>
<tr>
<th>Size of Aggregate</th>
<th>Value Returned in</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>EAX-EDX pair</td>
</tr>
<tr>
<td>5, 6, 7 bytes</td>
<td>EAX. The address to place return values is passed as a hidden parameter in EAX.</td>
</tr>
<tr>
<td>4 bytes</td>
<td>EAX</td>
</tr>
</tbody>
</table>
CDECL linkage

3 bytes  EAX  The address to place return values is passed as a hidden parameter to EAX.

2 bytes  AX

1 byte   AL
For functions that return aggregates 5, 6, 7 or more than 8 bytes in size, the address to place the return values is passed as a hidden parameter, and the address is passed back in EAX.

- Function names are decorated with an underscore prefix when they appear in object modules. For example, a function named fred in the source program will appear as _fred in the object.

When building export or import lists in .DEF files, the decorated version of the name should be used. If you used undecorated names in the DEF file, you must give the object files to ILIB along with the DEF file. ILIB uses the object files to determine how each name ended up after decoration.

Examples using the CDECL convention
The following examples are included for purposes of illustration and clarity only. They have not been optimized. The examples assume that you are familiar with programming in assembler. In the examples, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

Consider the following call:
m = func(a,b,c);

The variables a, b, and c are 32-bit integers and FUNC has two local variables, x and y (both 32-bit integers).

The stack for the call to FUNC would look like this:

```
<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Higher Memory

EDP ─────/SM590000 └───────────────────────┘ /SM630000─┘

These would only be pushed if they were used in this function.

Lower Memory
CDECL linkage

The instructions used to create this activation record on the stack look like this on the calling side:

```assembly
PUSH c
PUSH b
PUSH a
CALL _func
.
.
ADD ESP, 12 : cleaning up the parameters
.
.
MOV m, EAX
.
.
```

For the callee, the code looks like this:

```assembly
_func PROC
PUSH EBP
MOV EBP, ESP ; Allocating 8 bytes of storage
SUB ESP, 08H ; for two local variables.
PUSH EDI ; These would only be
PUSH ESI ; pushed if they were used
PUSH EBX ; in this function.
.
MOV EAX, [EBP - 8] ; Load y into EAX
MOV EBX, [EBP + 12] ; Load b into EBX
.
.
XOR EAX, EAX ; Zero the return value
POP EBX ; Restore the saved registers
POP ESI
POP EDI
LEAVE ; Equivalent to MOV ESP, EBP
; POP EBP
RET
_func ENDP
```

The saved register set is EBX, ESI, and EDI. In the case where the structure is passed as a value parameter and the size of the structure is 5, 6, 7, or more than 8 bytes in size, the address to place the return values is passed as a hidden parameter, and the address passed back in EAX.
Chapter 22. Using PL/I in mixed-language applications

Within the workstation environment, there are occasions when you want to develop mixed-language applications with PL/I being one of the languages involved. For example, an application could be constructed with the main program written in C and a dynamic link library (DLL) written in PL/I. Another possibility is an application using REXX which can load and call PL/I routines packaged in a PL/I DLL.

Perhaps you want to construct an application using software from an outside vendor. Using a vendor’s prepackaged program, you can supply a user exit in the form of a DLL written in PL/I.

Creating mixed-language applications is generally challenging and you have to consider many factors that do not exist when coding in a single language. Typically, high level programming languages from different vendors (for example, C, C++, COBOL, and PL/I) require the use of specific run-time environments as implemented by the run-time libraries of the distinct languages. Areas in which these languages might not work well together include:

- Implementations and usages of data types
- Data alignments
- Exception handling facilities
- Run-time environment initialization and termination
- User exit routines
- Input and output facilities

These inconsistencies in behavior can cause unexpected run-time behavior that can arise in some mixed-language program execution scenarios.

Matching data and linkages

For any routine to invoke another routine successfully, the two routines should have matching views of shared interfaces. When one of the routines is not coded in PL/I, these interfaces are limited by

- What data is passed
- How data is passed
- Where data is passed

The sections that follow describe these situations in more detail. Mismatched views of shared interfaces is a common problem in mixed language applications. Important points to remember are:

- Arguments and parameters must match
- Data that is meant to be received by value should be passed by value
- Both the called and calling routines should use the same linkage.

What data is passed

PL/I and C routines communicate by passing and returning data of equivalent data types. PL/I and non-PL/I routines should not communicate by using external static variables. Table 29 on page 412 lists the scalar data types which are equivalent between PL/I and C.
### Matching data and linkages

Table 29. Equivalent data types between C and PL/I

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>PL/I Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>FIXED BIN(7,0)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>UNSIGNED FIXED BIN(8,0) or CHAR(1)</td>
</tr>
<tr>
<td>signed short</td>
<td>FIXED BIN(15,0)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>UNSIGNED FIXED BIN(16,0)</td>
</tr>
<tr>
<td>signed (long) int</td>
<td>FIXED BIN(31,0)</td>
</tr>
<tr>
<td>unsigned (long) int</td>
<td>UNSIGNED FIXED BIN(31,0)</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT BIN(21) FLOAT DEC(6)</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT BIN(53) FLOAT DEC(16)</td>
</tr>
<tr>
<td>long double</td>
<td>FLOAT BIN(64) FLOAT DEC 18)</td>
</tr>
<tr>
<td>enum</td>
<td>ORDINAL</td>
</tr>
<tr>
<td>&lt;non-function-type&gt; *</td>
<td>POINTER or HANDLE</td>
</tr>
<tr>
<td>&lt;function-type&gt; *</td>
<td>ENTRY LIMITED</td>
</tr>
</tbody>
</table>

As is illustrated in the last row of the table, a C function pointer is not equivalent to a PL/I entry variable unless the entry variable is LIMITED. Errors caused by this mistake are hard to detect.

Arrays of equivalent types are equivalent as long as they have the same number of dimensions and the same lower and upper bounds. In C, you cannot specify lower bounds, and the actual upper bound is one less than the number you specify. For example, consider this array declared in C:

```c
short x[6];
```

In PL/I, the array would be declared as follows:

```pl/i
dc1 x(0:5) fixed bin(15);
```

Structures and unions of equivalent types are also equivalent if their elements are mapped to the same offsets. The offsets are the same if there is no padding between elements. If the elements of a structure (or union) are all UNALIGNED, PL/I does not use padding. When some elements are ALIGNED, you can determine if there is any padding by examining the AGGREGATE listing. PL/I regards strings as scalars but C does not; therefore, none of the previous discussion applies to strings.

C bit fields have only nominal resemblance to PL/I bit strings.

- C bit fields are limited to 32 bits, but PL/I bit strings can be as long as 32767 bits.
- C bit fields are not always mapped in left-to-right order. Some Intel C compilers would map the following C structure so that it is equivalent to the PL/I structure:

  **C Structure**
  ```c
  struct { unsigned byte0 :8;
          unsigned byte1 :8;
          unsigned byte2 :8;
          unsigned byte3 :8;
        } bytes;
  ```

  **PL/I Structure**

  ```pl/i
  ```
Matching data and linkages

dcl
  1 bytes,
  2 byte1 bit(8),
  2 byte2 bit(8),
  2 byte3 bit(8),
  2 byte4 bit(8);

Other C compilers would map the original structure with the bytes reversed so that it would be equivalent to this PL/I structure.

**PL/I Structure**

dcl
  1 bytes,
  2 byte4 bit(8),
  2 byte3 bit(8),
  2 byte2 bit(8),
  2 byte1 bit(8);

Strictly speaking, C has no character strings, but only pointers to char. However, by common usage, a C string is a sequence of characters the last of which has the value '00'X. Thus, in the example below, *address* is a C 'string' that could hold up to 30 non-null characters.

```c
char address[31];
```

The following PL/I declare most closely resembles the C 'string'.

```pli
dcl address char(30) varyingz;
```

In the declarations of C functions, strings are usually declared as char*. For example, the C library function *strcspn* could be declared as:

```c
int strcspn(char* string1, char* string2);
```

The PL/I declare for the same function would be:

```pli
dcl strcspn entry(char(*) varyingz,
                          char(*) varyingz)
               returns(fixed bin(31));
```

In the preceding examples, both the C and PL/I declarations are incomplete. Complete versions are given and explained later in this chapter.

**How data is passed**

Both PL/I and C support various methods of passing data. To understand these methods, you must know the following terms:

**Parameter**

A variable declared in a PL/I procedure or function definition. For example, *seed* is a parameter in the following PL/I function definition.

```pli
funky:
  proc(seed)
  returns(fixed bin(31));

dcl seed fixed bin(31);
  .
  .
end funky;
```

**Argument**

A variable or value actually passed to a routine. When the function *funky* (from the preceding example) is invoked by `rc = funky(seed);`, *seed* is an argument.
Matching data and linkages

By value
The value of the argument is passed. When a calling routine passes an argument by value, the called routine cannot alter the original argument.

By address
The address of the argument is passed. When a calling routine passes an argument by address, the called routine can alter the caller’s argument.

C passes all parameters by value, but PL/I (by default) passes parameters by address. PL/I also supports passing parameters by value except for arrays, structures, unions, and strings with length declared as *.

As is described in more detail in the PL/I Language Reference, you can indicate if a parameter is passed by address or by value by declaring it with the BYADDR or BYVALUE attribute. In the following example, the first parameter to modf is passed by value, while the second is passed by address.

```pli
dcl modf entry( float bin(53) byvalue,
               float bin(53) byaddr )
returns( float bin(53) );
```

The corresponding C declaration is:

```c
double modf( double x, double * intptr );
```

If the BYADDR or BYVALUE attributes are not explicit in the declaration, you can specify them in the options list for that entry. The following declare uses the options list making it equivalent to the previous example.

```pli
dcl modf entry( float bin(53),
               float bin(53) byaddr )
returns( float bin(53) )
options( byvalue );
```

Even when a parameter is passed by address, its value might not be changed by the receiving routine. You can indicate this in PL/I by adding the attribute NONASSIGNABLE (or NONASGN) to the declaration for that parameter. The following partial declaration indicates that neither of the arguments to the function strcspn is altered by that function:

```pli
dcl strcspn entry( nonasgn char(*) varyingz,
                   nonasgn char(*) varyingz )
returns( fixed bin(31) );
```

The corresponding C declaration is:

```c
int strcspn( const char * string1, const char * string2 );
```

A routine must agree with any routines that call it about how data is passed between them. You can avoid potential problems by giving the compiler enough information to detect these kinds of mismatches. For example, while the following declare is technically equivalent to the declare for modf in the sample code shown earlier, it allows the address of any argument to be passed as the second argument. The earlier declares would require the second argument to have the correct type.

```pli
dcl modf entry( float bin(53),
                pointer )
returns( float bin(53) )
options( byvalue );
```

Finally, when PL/I passes some data types (strings, arrays, structures, and unions), it also, by default, passes a descriptor that describes data extents (maximum string length, array bounds, etc.). Since C routines cannot consume PL/I descriptors, you
Matching data and linkages

should keep descriptors from being passed between C and PL/I routines. You can
do this by adding the NODESCRIPTOR option to the OPTIONS attribute in the
declaration for the C entry, for example:

    dcl strncspn entry( nonasgn byaddr char(*) varyingz,
               nonasgn byaddr char(*) varyingz )
         returns( fixed bin(31) )
    options( nodescriptor );

Where data is passed

It is as important for interacting routines to agree on what and where data is
passed as it is for them to agree on how data is passed. With both PL/I and C,
data can be passed on the stack, in general registers, or in floating-point registers.

In PL/I, the LINKAGE option (in the OPTIONS option of the procedure statement
and entry declaration) determines where data is passed. One common way that
effects in data location occur is if you specify mismatched linkage types (or fail to
specify a linkage type when the default is incorrect).

PL/I for Windows supports three 32-bit linkage types— OPTLINK, CDECL and
STDCALL. The following PL/I declaration indicates that the function dosSleep uses
the SYSTEM linkage:

    dcl dosSleep entry( fixed bin(31) byvalue )
         returns( fixed bin(31) )
    options( linkage(system) );

The options list should specify the linkage used by any C routines you call. Both
the PL/I and VisualAge for C++ compilers use OPTLINK as their default linkage.
Many C routines on Windows use the STDCALL linkage, and for these routines,
LINKAGE(STDCALL) should be specified in the OPTIONS attribute. For instance,
you would declare the Windows equivalent of DosSleep as:

    dcl Sleep ext('Sleep')
     entry( fixed bin(31) byvalue )
         returns( fixed bin(31) )
    options( linkage(stdcall) );

Maintaining your environment

In order for PL/I (and many other languages) to work correctly, you must not
damage the runtime environment they establish. When interlanguage calls are
involved, this means that:

• Any routine that registers an exception handler should deregister that handler
before returning to PL/I

• Out-of-block GOTOs are permitted only if the source and target blocks are coded
in the same language and any intervening blocks are coded in the same
language.

Invoking non-PL/I routines from a PL/I MAIN

If your main routine is programed in PL/I, you can call two kinds of non-PL/I
routines:

• System routines (such as DOS and Windows services)
• C or COBOL

System routines do not require their own runtime environment, and they can be
linked directly into a PL/I executable (.EXE) file or dynamic link library (.DLL).
With the exception of IBM VisualAge C/C++ routines, all other non-PL/I routines
**Invoking non-PL/I routines**

should **not** be linked directly into an .EXE or .DLL. They should be linked instead into a .DLL so that any run-time environment initialization that they require can be performed when that .DLL is loaded.

IBM VisualAge C/C++ routines can be linked with PL/I. However, if C routines are linked with PL/I and any of them use C library functions (or are C library functions themselves), the C runtime must be initialized before any routines are called. The C runtime can be initialized by calling the following routine

```c
dcl _CRT_init ext('_CRT_init')
  entry()
  returns( optional fixed bin(31) )
  options( linkage(optlink) );
```

Also, in order to ensure that the C runtime closes all files it opened and returns any other system resources it may have acquired, you have to terminate the C runtime by calling

```c
dcl _CRT_term ext('_CRT_term')
  entry()
  returns( optional fixed bin(31) )
  options( linkage(optlink) );
```

**Invoking PL/I routines from a non-PL/I main**

The PL/I run-time environment has the ability to:

- Self-initialize when a PL/I DLL is dynamically loaded from a non-PL/I main program.
- Exist with a non-PL/I language run-time environment with minimal conflicts.

A PL/I routine invoked from a non-PL/I routine should handle any exceptions that occur in PL/I code and returns to the non-PL/I using a RETURN or END statement in the first PL/I procedure (see "Using ON ANYCONDITION").

The PL/I run-time implicitly frees any resources acquired by PL/I, but not until the application terminates.

You can also explicitly release resources through various PL/I statements:

- RELEASE * - releases all fetched modules
- FLUSH FILE(*) - flushes all file buffers
- CLOSE FILE(*) - closes all open files

**Using ON ANYCONDITION**

Any application should be able to handle all exceptions that occur within it and return 'normal' control to the calling program. PL/I exception-handling facilities and ANYCONDITION ON-units help make this possible.

The first executable statement in any PL/I routine that is called from a non-PL/I routine should be an ON ANYCONDITION statement. This statement should contain code to handle any condition not handled explicitly by other ON-units. If a condition arises that cannot be handled, use a GOTO statement pointing to the last statement that would normally be executed in the routine, for example:

```pli
pliapp:  
  proc( p1, ..., pn )
  returns( ... )
  options( fromalien );
```
Invoking PL/I routines

/* declarations of parameters, if any */
/* declarations of other variables */
on anycondition
    begin;
        /* handle condition if possible */
        /* if unhandled, set return value */
        goto return_stmt;
    end;
/* mainline code */

returnStmt:
    return(...);

end_stmt:
    end pliapp;

For PL/I routines that are not functions, the target for the GOTO should be the END statement in the routine.
Chapter 23. 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. A simple Java - PL/I application will be described and information on compatibility between the two languages will also be discussed.

Before you can communicate with Java from PL/I you need to have Java installed on your system. There are many places to download a free version of the latest Java Development Kit (JDK).

What is the 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 the JNI, you 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 native 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 would do this from Java with a method call to a native language, known as a native method.

Some reasons to use native methods may include the following:
- 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 wish to implement a intensive series of complicated calculations in your native language and have your Java applications call these functions.
- 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.
- create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks.
- inspect and use objects created by Java application code.
- 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.

**JNI Sample Program #1 - 'Hello World'**

The first sample program we will write is yet another variation of the "Hello World!" program.

Our "Hello World!" program has one Java class, `callingPLI.java`. Our native method, written in PL/I, is contained in `hiFromPLI.pli`. Here is a brief overview of the steps for creating 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 which calls the native method in the PL/I program.

### Step 1: Writing the Java Program

**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. The declaration of our native method looks like this:

```java
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.

**Load the Native Library**

A step that loads the native library must be included so the native library will be loaded at execution time. The Java statement that loads the native library looks like this:

```java
static {
    System.loadLibrary("hiFromPLI");
}
```

In the above statement, the Java System method `System.loadLibrary(...)` is called to find and load the dynamic link library (DLL). The PL/I dynamic link library, `hiFromPLI.dll`, will be created during the step that compiles and links the PL/I program.

**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 section, looks like this:
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
Use the Java compiler to compile the callingPLI class into an executable form. The command would look like this:
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(31) ) Margins(1, 100);
*Process Dllinit xinfo(def);
*Process Default(IEEE);

Here is a brief description of them and why they are useful:

Extname(31)
    Allows for longer, Java style, external names.

Margins(1,100)
    Extending the margins gives you more room for Java style names and identifiers.

Dllinit
    Includes the initialization coded needed for creating a DLL.

xinfo(def)
    Instructs the compiler to build a *.DEF file to be used in the creation of the DLL.

Default(IEEE);
    IEEE specifies that FLOAT data is held in IEEE format - the form in which it is held by JAVA

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 the external 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 shipped 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.

The complete procedure statement for the sample program looks like this:

```pli
Java_callingPLI_callToPLI:
  Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( NoDescriptor ByValue linkage(stdcall) );
```

**JNI Include File**
The PL/I include file which contains the PL/I definition of the Java interfaces is contained in two include files, jni.cpy which in turn includes jni_md.cpy. These include files are included with this statement:

```pli
%include jni;
```

For a complete listing of the jni.cpy file look in the \ibmpliw\include directory

**The Complete PL/I Procedure**
For completeness, here is the entire PL/I program that defines the native method:

```pli
*Process Limits( Extname( 31 ) ) Margins( 1, 100 ) ;
*Process Dllinit xinfo(def);
*Process Default( IEEE );
PliJava_Demo: Package Exports(*);

Java_callingPLI_callToPLI:
  Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( NoDescriptor ByValue linkage(stdcall) );

  %include jni;

  Display('Hello from PL/I for Windows!');

End;
```

**Step 4: Compiling and Linking the PL/I Program**

**Compiling the PL/I Program**
Compile the PL/I sample program with the following command:

```pli
pli hiFromPLI.pli
```

**Linking the Dynamic Link Library**
Link the resulting PL/I object deck into a DLL with these commands:

```pli
ilb /nologo /geni hiFromPLI.def
ilink /dll hiFromPLI.obj hiFromPLI.exp
```
Step 5: Running the Sample Program
Run the Java - PL/I sample program with this command:
```
java callingPLI
```

The output of the sample program will look like this:
```
Hello from PL/I for Windows!
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. Refer to Figure 30 on page 424 for the complete listing of the jPassString.java program. The Java portion has one Java class, jPassString.java. Our native method, written in PL/I, is contained in passString.pli. Much of the information from the first sample program applies to this sample program as well. Only new or different aspects will be discussed for this sample program.

Step 1: Writing the Java Program

Declare the Native Method
The native method for this sample program looks like this:
```
public native void pliShowString();
```

Load the Native Library
The Java statement that loads the native library for this sample program looks like this:
```
static {
    System.loadLibrary("passString");
}
```

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 30 on page 424.
Step 2: Compiling the Java Program

The command to compile the Java code would look like this:

`javac jPassString.java`
Step 3: Writing the PL/I Program

All of the information about writing the PL/I "Hello World" sample program applies to this program as well.

Correct Form of PL/I Procedure Name and Procedure Statement

The external PL/I procedure name for this program would be

```
_java_jPassString_pliShowString
```

The complete procedure statement for the sample program looks like this:

```
Java_jPassString_pliShowString:
Proc( JNIEnv , myjobject )
   external( "_Java_jPassString_pliShowString" )
   options( byvalue nodelist stdcall )
```

JNI Include File

The PL/I include file which contains the PL/I definition of the Java interfaces is contained in two include files, `jni.cpy` which in turn includes `jni_md.cpy`. These include files are included with this statement:

```
%include jni;
```

For a complete listing of the `jni.cpy` file look in the `ibmpliw\include` directory

The Complete PL/I Procedure

The complete PL/I program is shown in Figure 31 on page 426. 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 will use this reference to get information from the calling object. The first piece of information is the Class of the calling object which is retrieved using 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 that we are interested in. This Java field is further identified by providing 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 using 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 is used to convert 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 using the `NewString` JNI function. This new Java string is then used to update the string field in the calling Java object using 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.
Step 4: Compiling and Linking the PL/I Program

Compiling the PL/I Program

Compile the PL/I sample program with the following command:

```
pli passString.pli
```
Linking the Dynamic Link Library

Link the resulting PL/I object deck into a DLL with these commands:

```
ilib /nologo /geni passString.def
ilink /dll passString.obj passString.exp
```

Step 5: Running the Sample Program

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, will look like this:

```
>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. Refer to Figure 32 on page 428 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 the first sample program applies to this sample program as well. Only new or different aspects will be discussed for this sample program.

Step 1: Writing the Java Program

Declare the Native Method

The native method for this sample program looks like this:

```
public native void pliShowInt();
```

Load the Native Library

The Java statement that loads the native library for this sample program looks like this:

```
static {
    System.loadLibrary("passInt");
}
```

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 32 on page 428.

---

Chapter 23. Interfacing with Java  427
// 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(" ");
        jPassInt pInt = new jPassInt();
        pInt.myInt = 1024;
        pInt.myString = "";
        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) {
            /* Say something bad happened */
            System.out.println("Something bad happened... ");
        }
    }
}

Figure 32. Java Sample Program #3 - Passing an Integer
Step 2: Compiling the Java Program
The command to compile the Java code would look like this:

```
javac jPassInt.java
```

Step 3: Writing the PL/I Program
All of the information about writing the PL/I "Hello World" sample program applies to this program as well.

Correct Form of PL/I Procedure Name and Procedure Statement
The external PL/I procedure name for this program would be 

```
 Java_jPassInt_pliShowInt
```

The complete procedure statement for the sample program looks like this:

```
Java_passNum_pliShowInt:
Proc( JNIEnv , myjobject )
   external( "Java_jPassInt_pliShowInt" )
   options( byvalue nodescriptor linkage(stdcall) );
```

JNI Include File
The PL/I include file which contains the PL/I definition of the Java interfaces is contained in two include files, `jni.cpy` which in turn includes `jni_md.cpy`. These include files are included with this statement:

```
%include jni;
```

For a complete listing of the `jni.cpy` file look in the `ibmplied\include` directory

The Complete PL/I Procedure
The complete PL/I program is shown in Figure 33 on page 430. 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 will use this reference to get information from the calling object. The first piece of information is the Class of the calling object which is retrieved using 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 that we are interested in. This Java field is further identified by providing 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 using 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 using 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.
Step 4: Compiling and Linking the PL/I Program

### Compiling the PL/I Program

Compile the PL/I sample program with the following command:

```
pli passInt.pli
```

### Linking the Dynamic Link Library

Link the resulting PL/I object deck into a DLL with these commands:

```
ilib /nologo /geni passInt.def
ilink /dll passInt.obj passInt.exp
```
**Step 5: Running the Sample Program**

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, will look like this:
```
> java jPassInt

From Java: Enter an Integer or 'quit' to quit.
Java Prompt > 12345

From PL/I: Integer retrieved from Java is: 12345
From PL/I: 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
```

**Determining equivalent Java and PL/I data types**

When you communicate with Java from PL/I you will need to match the data types between the two programming languages. This table shows Java primitive types and their PL/I equivalents:

<table>
<thead>
<tr>
<th>Java Type</th>
<th>PL/I Type</th>
<th>Size in Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>jboolean</td>
<td>8, unsigned</td>
</tr>
<tr>
<td>byte</td>
<td>jbyte</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>jchar</td>
<td>16, unsigned</td>
</tr>
<tr>
<td>short</td>
<td>jshort</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>jint</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>jlong</td>
<td>64</td>
</tr>
<tr>
<td>float</td>
<td>jfloat</td>
<td>21</td>
</tr>
<tr>
<td>double</td>
<td>jdouble</td>
<td>53</td>
</tr>
<tr>
<td>void</td>
<td>jvoid</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Chapter 24. Using sort routines

PL/I for Windows supports the PLISRTx (x = A, B, C, or D) built-in subroutines. To use the PLISRTx subroutines, you need to:

- Include a call to one of the subroutines and pass it the information on the fields to be sorted. This information includes the length of the records, the name of a variable to be used as a return code, and other information required to carry out the sort.
- Specify the data sets required by the sort program in DD statements.

When used from PL/I, these subroutines sort 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 PL/I procedure written by the programmer that the sort program calls 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.

Comparing S/390 and workstation sort programs

If your existing mainframe programs contain CALL PLISRTx, you can download and run them on your workstation. Several of the parameters allowed on S/390 are ignored, and alter run-time behavior to some extent. The following table indicates which arguments accepted by OS PL/I are ignored by the workstation compiler.

Table 31. workstation PLISRTx

<table>
<thead>
<tr>
<th>Built-in subroutine</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISRTA</td>
<td>(sort statement, record statement, storage, return code, [data set prefix, message level, sort technique])</td>
</tr>
<tr>
<td>Sort input: data set</td>
<td></td>
</tr>
<tr>
<td>Sort output: data set</td>
<td></td>
</tr>
<tr>
<td>PLISRTB</td>
<td>(sort statement, record statement, storage, return code, input routine, [data set prefix, message level, sort technique])</td>
</tr>
<tr>
<td>Sort input: PL/I subroutine</td>
<td></td>
</tr>
<tr>
<td>Sort output: data set</td>
<td></td>
</tr>
<tr>
<td>PLISRTC</td>
<td>(sort statement, record statement, storage, return code, output routine, [data set prefix, message level, sort technique])</td>
</tr>
<tr>
<td>Sort input: data set</td>
<td></td>
</tr>
<tr>
<td>Sort output: PL/I subroutine</td>
<td></td>
</tr>
<tr>
<td>PLISRTD</td>
<td>(sort statement, record statement, storage, return code, input routine, output routine, [data set prefix, message level, sort technique])</td>
</tr>
<tr>
<td>Sort input: PL/I subroutine</td>
<td></td>
</tr>
<tr>
<td>Sort output: PL/I subroutine</td>
<td></td>
</tr>
</tbody>
</table>
Comparing sort programs

Table 31. workstation PLISRTx (continued)

Built-in subroutine | Argument definitions:
--- | ---
Sort statement | Character string expression describing sorting fields and format. See “Specifying the sorting field” on page 437.
Record statement | Character string expression describing the length and record format of data. See “Specifying the records to be sorted” on page 438.
Storage | Ignored by workstation PL/I.
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
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 15. For specific requirements using workstation PL/I, see “E15 — input-handling routine (sort exit E15)” on page 442.
Output routine | (PLISRTC and PLISRTD only.) Name of the PL/I external or internal procedure to which Sort passes the sorted records from sort exit 35. For specific requirements using workstation PL/I, see “E35 — output-handling routine (sort exit E35)” on page 444.
Data set prefix | Ignored by workstation PL/I, which only processes SORTIN and SORTOUT as ddnames.
Message level | Ignored by workstation PL/I.
Sort technique | Ignored by workstation PL/I.

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, and the length of your data records.

To determine which PLISRTx built-in subroutine to 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 the data before it is sorted, and to make immediate use of the data in its sorted form. If you decide to use an input or output handling subroutine, read “Sort data handling routines” on page 441.

The sort built-in subroutines and the source and destination of data are as follows:

<table>
<thead>
<tr>
<th>Built-in subroutine</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISRTA</td>
<td>Data set</td>
<td>Data set</td>
</tr>
<tr>
<td>PLISRTB</td>
<td>Subroutine</td>
<td>Data set</td>
</tr>
<tr>
<td>PLISRTC</td>
<td>Data set</td>
<td>Subroutine</td>
</tr>
</tbody>
</table>
Preparing to use sort

<table>
<thead>
<tr>
<th>Built-in subroutine</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISRTD</td>
<td>Subroutine</td>
<td>Subroutine</td>
</tr>
</tbody>
</table>

Source data sets are defined using the SORTIN environment variable while destination data sets are defined using SORTOUT. Alternatively, you can use the PUTENV built-in function to set those functions.

Having determined the subroutine you are using, you must now determine a number of things about your data set and specify the information on the SORT statement:

- 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.

Next, you must determine two things about the records to be sorted and specify the information on the RECORD statement:

- Whether the record format is fixed or varying
- The length of the record (maximum length for varying)

You use these on the RECORD statement, which is the second argument to PLISRTx.

Choosing the type of sort

To make the best use of the sort program, you should understand how it works. In your PL/I program you specify a sort by using a CALL statement to the built-in subroutine PLISRTx. 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 PLISRTx 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 name of a variable into which sort places 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 from the information supplied by the PLISRTx argument list and depends on your choice of A, B, C, or D for x. Control is then transferred to the sort program. If you have specified an output- or input-handling routine, it is called by the sort program as many times as is necessary to handle each of the unsorted or sorted records.

The sort operation ends in one of two ways:

1. Communicating success or failure by sending a return code to the PL/I calling procedure.
2. Raising an error condition when certain errors are detected and the return code is undefined.

Figure 34 on page 436 is a simplified flowchart showing the sort operation.
Preparing to use sort

Within the sort program itself, the flow of control between the sort program and output- and input-handling routines is controlled by return codes. The sort program calls these routines at the appropriate point in its processing. (Within the sort program, 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, the sort program expects a return code indicating either that it should call the routine again, or that it should continue with the next stage of processing.

Figure 34. Flow of control for the sort program

Within the sort program itself, the flow of control between the sort program and output- and input-handling routines is controlled by return codes. The sort program calls these routines at the appropriate point in its processing. (Within the sort program, 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, the sort program expects a return code indicating either that it should call the routine again, or that it should continue with the next stage of processing.
Preparing to use sort

The remainder of this chapter gives detailed information on how to use sort from PL/I. First the required PL/I statements are described, followed by the data set requirements. The chapter finishes with a series of examples showing the use of the four built-in subroutines.

Specifying the sorting field

The SORT statement is the first argument to PLISRTx. The syntax of the SORT statement must be a character string expression that takes the form:

```
'SORTbFIELDS=(start,length,form,seq)'
```

- **b** One or more blanks. Blanks shown are mandatory. No other blanks are allowed.
- **start**, **length**, **form**, **seq**
  - Sorting fields. You can specify any number of such 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 is preserved. The overlaying of sort fields is not supported.
  - **start**
    - The starting position within the record. Give the value in bytes. The first byte in a string is considered to be byte 1.
  - **length**
    - The length of the sorting field. Give the value in bytes. The length of sorting fields is restricted according to their data type.
  - **form**
    - 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Data Type and Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>character 1–65535</td>
</tr>
<tr>
<td>ZD</td>
<td>zoned decimal signed 1–255</td>
</tr>
<tr>
<td>PD</td>
<td>packed decimal signed 1–255</td>
</tr>
<tr>
<td>BI</td>
<td>binary, unsigned 1 bit to 4095 bytes</td>
</tr>
</tbody>
</table>

- **Note:** If the file is EBCDIC, the PL/I library changes the sort argument from CH to BI to use the binary collating sequence for the field specified. The binary field limit is 4095.

- **seq**
  - The sequence in which the data is sorted:
    - A – ascending (that is, 1,2,3,...)
    - D – descending (that is, ...,3,2,1).
  - Note that you cannot specify E, because PL/I does not provide a method of passing a user-supplied sequence.
Preparing to use sort

**other options**
The only option supported under workstation PL/I is the default, EQUALS. Source code downloaded from the mainframe, however, does not need to be altered.

**Example:**
```
' SORT FIELDS=(1,10,CH,A) '  
```

**Specifying the records to be sorted**

Use the RECORD statement as the second argument to PLISRTx. The syntax of the RECORD statement must be a character string expression which, when evaluated, accepts the following syntax:

```
' RECORD TYPE=rectype,LENGTH=(n) '  
```

- **b** 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

Even when you use input and output routines to handle the sorted and unsorted 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 is taken from the input data set. The maximum length of a record that can be sorted is 32,767 bytes. For varying-length records, you must include the 2-byte prefix.

- **n** The length of the record to be sorted.

**Note:** Additional length specifications that can be used are ignored by workstation PL/I.

**Example:**
```
' RECORD TYPE=F,length=(80) '  
```

**Calling the sort program**

When you have determined the sort field and record type specifications, you are in a position to write the CALL PLISRTx statement.

**PLISRT examples**

The following examples indicate commonly used forms of calls to PLISRTx.

**Example 1**

A call to PLISRTA sorting 80-byte records from SORTIN to SORTOUT, and a return code, RETCODE, declared as FIXED BINARY (31,0).
Calling the sort program

call plisrta (' SORT FIELDS=(1,80,CH,A) ',
               ' RECORD TYPE=F,LENGTH=(80) ',
               0,
               retcode);

Example 2
This example is the same as example 1 but 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) ',
               0,
               retcode);

Example 3
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. Other information as above.

call plisrtb (' SORT FIELDS=(1,10,CH,A) ',
               ' RECORD TYPE=F,LENGTH=(80) ',
               0,
               retcode,
               putin);

Example 4
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 82 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. If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

call plisrtdd (' SORT FIELDS=(1,5,CH,A,6,5,CH,D),EQUALS ',
                ' RECORD TYPE=V,LENGTH=(82) ',
                0,
                retcode,
                putin,       /* input routine (sort exit 15) */
                putout);     /* output routine (sort exit 35) */

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 PLISRTx. It then returns control to the statement that follows the CALL PLISRTx statement. The value returned indicates the success or failure of the sort as follows:

0   Sort successful
16  Sort failed

You must declare this variable as FIXED BINARY (31,0). It is standard practice to test the value of the return code after the CALL PLISRTx statement and take appropriate action according to the success or failure of the operation.

For example (assuming the return code was called RETCODE):

if retcode=0 then do;
   put data(retcode);
   signal error;
end;
**Calling the sort program**

The error condition is raised if errors are detected. When sort detects a fatal error and the corresponding error code is greater than 16, the error condition is raised.

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. The following example shows how you can call PLIRETC with the value returned from sort:

\[
\text{call pliretc(retcode);}
\]

You should not confuse this call to PLIRETC with the calls made in the input (E15) and output (E35) routines, where a return code is used for passing control information to sort.

---

**Getting diagnostic information**

This topic describes environment variables that you can specify to get helpful diagnostic information.

**The IBM_SORT_RC environment variable**

When you set the value of the IBM_SORT_RC environment variable to 1, the PL/I library returns more informative return codes from the PLISRTx calls, for example:

\[
\text{export IBM_SORT_RC=1}
\]

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Type of Error Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-32</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>34</td>
<td>Read error on file detected.</td>
</tr>
<tr>
<td>35</td>
<td>Write error on file detected.</td>
</tr>
<tr>
<td>36</td>
<td>Cannot open input file.</td>
</tr>
<tr>
<td>37</td>
<td>Cannot open message file.</td>
</tr>
<tr>
<td>38</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>39</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>40</td>
<td>Not enough temporary disk space.</td>
</tr>
<tr>
<td>43-47</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>49</td>
<td>Insufficient storage to continue execution.</td>
</tr>
<tr>
<td>50-55</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>57</td>
<td>Cannot open output file.</td>
</tr>
<tr>
<td>58</td>
<td>Cannot open temporary file.</td>
</tr>
<tr>
<td>61-62</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>112</td>
<td>No msg disk full.</td>
</tr>
<tr>
<td>232-233</td>
<td>Internal error detected.</td>
</tr>
<tr>
<td>254-255</td>
<td>Internal error detected.</td>
</tr>
</tbody>
</table>
The IBM_SORT_DIR environment variable

When you specify the IBM_SORT_DIR environment variable, temporary work files are saved in the directory you have specified instead of the current directory, for example:

```bash
export IBM_SORT_DIR=/tmp
```

The IBM_SORT_MSGFILE environment variable

When the IBM_SORT_MSGFILE environment variable is set to a path or file name, SORT messages are written to the specified file, for example:

```bash
export IBM_SORT_MSGFILE=./msg.txt
```

These messages are internal SORT error messages, which provide more information about the problem.

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. An example of a PLISRTA program is in Figure 38 on page 446. Other interfaces require either the input-handling routine or the output-handling routine, or both.

To sort varying-length records, you first need to convert your data sets to TYPE(VARLS) format, and then use this TYPE(VARLS) file as input to the sort program. TYPE(VARLS) records have a 2-byte length field at the beginning, so the record size is actually two less than the length of the record. This means the record size you specify should be two less than the maximum record length for the file.

You can convert your data set to a TYPE(VARLS) file by writing a PL/I program that reads from the existing data file and writes to an output file declared as TYPE(VARLS).

Sort data handling routines

The input-handling and output-handling routines are called by sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRTx, they behave in the same way as ordinary internal procedures with respect to the scope of names. The input and output procedure names themselves must be known in the procedure that makes the call to PLISRTx.

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 do 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.
Sort data handling routines

**E15 — input-handling routine (sort exit E15)**

Input routines are normally used to process data in some way before it is sorted, such as printing it, (see [Figure 39 on page 447](#) and [Figure 41 on page 449](#)), or generating or manipulating 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, and a return code of 12, which means the record passed is to be included in the sort. SORT continues to call the routine until a return code of 8 is passed. This means that all records have already been passed, and SORT is not to call the routine again. If a record is returned when the return code is 8, it is ignored by SORT.

**Note:** You must compile the program that calls PLISRTB or PLISRTD with the same options (ASCII or EBCDIC; NATIVE or NONNATIVE; HEXADEC or IEEE) that you used to compile the E15 handling routine.

The data returned by the E15 routine must be a fixed or varying character string. 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 PLISRTx. However, you can specify F, in which case the string is 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. Examples of an input routine are given in [Figure 39 on page 447](#) and [Figure 41 on page 449](#).

In addition to the return codes of 12 (include current record in sort) and 8 (all records sent), SORT allows the use of a return code of 16. This ends the sort and sets a return code from SORT to your PL/I program of 16—sort failed.

It should be noted that a call to PLIRETC sets a return code that is passed by your PL/I program, and is available to any job steps that follow it. When an output handling routine has been used, it is a good practice to reset the return code with a call to PLIRETC after the call to PLISRTx 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 40 on page 448](#).
In addition, to code the input user exit routine, the explicit attributes of the E15 must be specified in the program unit that calls PLSRTx if E15 is not nested in that program unit.

Figure 35. Skeletal code for an input procedure

```plaintext
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 is normally
       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. */
    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 */
        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 of the input procedure */
```

Sort data handling routines

Chapter 24. Using sort routines 443
Sort data handling routines

```
plisort: proc options(main);
    dcl e15 entry returns(char(2000) varying);
    /* Code to do your processing goes here */
    call plisrtb(' SORT FIELDS=(5,10,CH,A) ')
        ' RECORD TYPE=V,LENGTH=(2000) ',
        0,
        retcode,
        e15);
    /* Code to do your processing goes here */
end plisort;
*PROCESS
E15: proc returns (char(2000) varying);
    /* Returns option must be used specifying
       length of data to be sorted, maximum length
       if varying strings are passed to sort. */
    dcl string char(2000) varying;
        /* A character string variable is normally
           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. */
    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 */
        /* 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 36. When E15 is external to the procedure calling PLISRTx

**E35 — output-handling routine (sort exit E35)**

You must compile the program that calls PLISRTC or PLISRTD with the same
options (ASCII or EBCDIC; NATIVE or NONNATIVE) that you used to compile
the E35 handling routine.

Output-handling routines are normally used for any processing that is necessary
after the sort. This could be to print the sorted data, as shown in Figure 40 on page 448.
Sort data handling routines

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 PLISRTx statement. There is no indication from sort 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 PLISRTx.

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 a return code of 4 to sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

The sort can be stopped by passing a return code of 16 to sort. This results in sort returning to the calling program with a return code of 16—sort failed.

The record passed to the routine by sort is a character string parameter. If you specified the record type as F in the second argument in the call to PLISRTx, you should declare the parameter with the length of the record. If you specified the record type as V, you should declare the parameter as adjustable, for example:

dcl string char(*);

Skeletal code for a typical output-handling routine is shown in Figure 37.

You should note that a call to PLIRETC sets a return code that is passed by your PL/I program, and is 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 PLISRTx 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 at the end of this chapter.

```pli
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 37. Skeletal code for an output-handling procedure
Calling PLISRTA

/*DESCRIPTION*/
/*Sorting from an input data set to an output data set*/
/*Use the following statements:*/
/*set dd:sortin=ex106.dat,type(crlf),lrecl(80)*/
/*set dd:sortout=ex106.out,type(crlf),lrecl(80)*/

ex106: proc options(main);
  dcl Return_code fixed bin(31,0);
  call plisrta ('SORT FIELDS=(7,74,CH,A)',
               'RECORD TYPE=F,LENGTH=(80)',
               0,
               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);
  other put skip edit
         ('Invalid sort return_code = ', Return_code) (a,f(2));
end /* Select */;
/*Set pl/i return code to reflect success of sort */
call pliretc(Return_code);
end ex106;

Figure 38. PLISRTA—Sorting from input data set to output data set

Content of EX106.DAT to be used with Figure 38
003329 HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886 BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077 ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334 HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872 HOME TAVERN, WESTLEIGH
000931 FOREST, IVER, BUCKS
Calling PLISRTB

```plaintext
/* DESCRIPTION */
/* Sorting from an input-handling routine to an output data set */
/* Use the following statements: */
/* */
/* set dd:sysin=ex107.dat,type(crlf),lrecl(80) */
/* set dd:sortout=ex107.out,type(crlf),lrecl(80) */
/* */
ex107: proc options(main);

dcl Return_code fixed bin(31,0);

call plisrtb (' SORT FIELDS=(7,74,CH,A) ',
               ' RECORD TYPE=F,LENGTH=(80) ',
               0,
               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);
other put skip edit ('Invalid return_code = ',Return_code)(a,f(2));

end /* Select */;
/* Set pl/i return code to reflect success of sort */
call pliretc(Return_code);

e15x: /* Input-handling routine gets records from the input
      stream and puts them before they are sorted */
proc returns (char(80));
dcl sysin file stream 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 */
   goto ende15;
end;

get file (sysin) edit (infield) (l);
   put skip edit (infield)(a(80)); /* Print input */
   call pliretc(12); /* Request sort to include current
                     record and return for more */
   return(Infield);
ende15: end e15x;
end ex107;
```

Figure 39. PLISRTB—Sorting from input-handling routine to output data set

Content of EX107.DAT to be used with Figure 39
Sort data handling routines

Calling PLISRTC

ex108: proc options(main);

  dcl Return_code fixed bin(31,0);

  call plisrtc (' SORT FIELDS=(7,74,CH,A) ',
               ' RECORD TYPE=F,LENGTH=(80) ',
               0,
               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);
    other put skip edit
       ('Invalid return_code = ', Return_code) (a,f(2));
  end /* Select */;

  call pliretc (return_code);

  e35x: /* Output-handling routine prints sorted records */
    proc (Inrec);
      dcl inrec char(*);
      put skip edit (inrec) (a);
      call pliretc(4); /* Request next record from sort */
    end e35x;
end ex108;

Figure 40. PLISRTC—Sorting from input data set to output-handling routine

Content of EX108.DAT to be used with Figure 40

003329HOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886HOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077HOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOKK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
Calling PLISRTD, example 1

```plaintext
/******************************************************************/
/*                                                                 /
/* DESCRIPTION                                                                 /
/* Sorting an input-handling to output-handling routine             /
/* Use the following statement:                                     /
/* set dd:sysin=ex109.dat,type(crlf),lrecl(80)                      /
/*                                                                 /
/******************************************************************/

ex109: proc options(main);
  dcl Return_code fixed bin(31,0);
  call plisrtd (' SORT FIELDS=(7,74,CH,A) ',
                ' RECORD TYPE=F,LENGTH=(80) ',
                0,
                Return_code,
                e15x,
                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);
    other put skip edit
       ('Invalid return_code = ', Return_code) (a,f(2));
  end /* select */;

  /* Set pl/i return code to reflect success of sort */
  call pliretc(Return_code);

  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 */
        goto ende15;
      end;

      get file (sysin) edit (infield) (l);
      put skip edit (infield)(a);
      call pliretc(12); /* Input to sort continues */
      return(Infield);
    ende15:
      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;
```

Figure 41. PLISRTD—Sorting input-handling routine to output-handling routine
Sort data handling routines

Contents of EX109.DAT and EX110.DAT used with Figure 41 on page 449 and Figure 42 on page 451

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
Calling PLISRTD, example 2

```plaintext
ex110: proc options(main);

  /*******************************************************************/
  /* */
  /* PLISRTD: sorting from an input-handling rtn to an */
  /* output-handling routine. Records are varying-length. */
  /* */
  /*******************************************************************/
  dcl rc fixed bin(31,0);

  call plisrtd(' SORT FIELDS=(7,4,CH,A) ',
             ' RECORD TYPE=V,LENGTH=(80) ',
             256000,
             rc,
             e15x,
             e35x );

  select( rc );
  when(0) put skip edit
    ('Sort complete return code = 0') (a);
  when(16) put skip edit
    ('Sort failed return code = 16') (a);
  other put skip edit
    ('Invalid return code = ', rc) (a,f(2));
  end;

  call pliretc(rc);

  e15x: proc returns( char(80) varying );
    dcl infield char(80) var;

    on endfile(sysin) begin;
      put skip(3) edit('End of sort program input. ',
                       'Sortout output should follow') (a);
      call pliretc(8);         goto ende15;
    end;

    get file(sysin) edit(infield) (l);
    put skip edit( infield ) (a);
    call pliretc(12);
    return(infield);
  ende15:
    end e15x;

  e35x: proc ( inrec );
    dcl inrec char(*);

    put skip edit(inrec) (a);
    call pliretc(4);

    end e35x;

  end ex110;

Figure 42. PLISRTD—Sorting input-handling routine to output-handling routine

Sort data handling routines

Chapter 24. Using sort routines 451
Sort data handling routines
Chapter 25. Using the SAX parser

The compiler provides an interface called PLISAXx (x = A or B) that provides you basic XML capability to PL/I. The support includes a high-speed XML parser, which allows programs to consume inbound XML messages, check them for well-formedness, and transform their contents to PL/I data structures.

The XMLCHAR built-in function provides support for XML generation.

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, etc. The application provides handlers to deal with the events reported by the parser. The Simple API for XML or SAX is an example of an industry-standard event-based API.

For a tree-based API (such as the Document Object Model or 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 non-standard interfaces.
- It supports XML files encoded in either Unicode UTF-16 or any of several single-byte code pages listed below.
- The parser is non-validating, but does partially check well-formedness. See section 2.5.10,

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.

The XML parser is non-validating, but does partially check for well-formedness errors, and generates exception events if it discovers any.
The PLISAXA built-in subroutine

The PLISAXA built-in subroutine allows you to invoke the XML parser for an XML document residing in a buffer in your program.

```
PLISAXA(e,p,x,n,\texttt{c})
```

- **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

Note that if the XML is contained in a CHARACTER VARYING or a WIDECHAR VARYING string, then the ADDRDATA built-in function should be used to obtain the address of the first data byte.

Also note that 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

The PLISAXB built-in subroutine allows you to invoke the XML parser for an XML document residing in a file.

```
PLISAXB(e,p,x,\texttt{c})
```

- **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.

The SAX event structure

The event structure is a structure consisting of 24 LIMITED ENTRY variables which point to functions that the parser will invoke for various "events".

The descriptions below of each event refer to the example of an XML document in Figure 43 on page 455. In these descriptions, the term "XML text" refers to the string based on the pointer and length passed to the event.
In the order of their appearance in this structure, the parser may recognize the following events:

**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 above example, 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" in the example above.

**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" in the example above.

**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 is the only event where XML text includes the delimiters. The example above 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, this would be the string "sandwich".
**attribute_name**

This event occurs for each attribute in an element start tag or empty element tag, after recognizing a valid name. The parser passes the address and length of the text containing the attribute name. The only attribute name in the example 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 "sandwich" example at the beginning of the section 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" in the example, as attribute_characters events, and the single character "'" as an attribute_predefined_reference event, described next.

**attribute_predefined_reference**

This event occurs in attribute values for the five pre-defined 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 ",d" or ",hh", where "d" and "h" represent decimal and hexadecimal digits, respectively. 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, in the above example, 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 this 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 may comprise several segments. For instance, the content of the "meat" element in the example 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.

current_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.

current_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" and "h" represent decimal and hexadecimal digits, respectively. 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, and 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 and closing comment delimiters, "<!-" and "-->", respectively. In the example, 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 pre-defined 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 pre-defined 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 functions/events that are different are:

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 WIDECCHAR encoded using Unicode UTF-16, or in CHARACTER encoded using one of the explicitly supported single-byte character sets listed below. 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 below, 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).

<table>
<thead>
<tr>
<th>CCSID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01047</td>
<td>Latin 1 / Open Systems</td>
</tr>
<tr>
<td>01140, 00037</td>
<td>USA, Canada, etc.</td>
</tr>
<tr>
<td>01141, 00273</td>
<td>Austria, Germany</td>
</tr>
<tr>
<td>01142, 00277</td>
<td>Denmark, Norway</td>
</tr>
<tr>
<td>01143, 00278</td>
<td>Finland, Sweden</td>
</tr>
<tr>
<td>01144, 00280</td>
<td>Italy</td>
</tr>
<tr>
<td>01145, 00284</td>
<td>Spain, Latin America (Spanish)</td>
</tr>
<tr>
<td>01146, 00285</td>
<td>UK</td>
</tr>
<tr>
<td>01147, 00297</td>
<td>France</td>
</tr>
<tr>
<td>01148, 00500</td>
<td>International</td>
</tr>
<tr>
<td>01149, 00871</td>
<td>Iceland</td>
</tr>
</tbody>
</table>
Supported ASCII code pages

<table>
<thead>
<tr>
<th>CCSID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00813</td>
<td>ISO 8859-7 Greek / Latin</td>
</tr>
<tr>
<td>00819</td>
<td>ISO 8859-1 Latin 1 / Open Systems</td>
</tr>
<tr>
<td>00920</td>
<td>ISO 8859-9 Latin 5 (ECMA-128, Turkey TS-5881)</td>
</tr>
</tbody>
</table>

Specifying the code page

If your document does not include an encoding declaration in the XML declaration, or does not have an XML declaration at all, the parser uses the encoding information provided by the PLISAX built-in subroutine call in conjunction 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. An example of an XML declaration that includes an encoding declaration is:

```xml
<?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.

Specify the encoding declaration as follows:

**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 upper or lower case):

<table>
<thead>
<tr>
<th>IBM_</th>
<th>CP</th>
<th>CCSID_</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-</td>
<td>CP</td>
<td>CCSID-</td>
</tr>
<tr>
<td>CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Using an alias**

You can use any of the following supported aliases (in any mixture of lower and upper case):

<table>
<thead>
<tr>
<th>Code page</th>
<th>Supported aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>037</td>
<td>EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL</td>
</tr>
<tr>
<td>500</td>
<td>EBCDIC-CP-BE, EBCDIC-CP-CH</td>
</tr>
<tr>
<td>813</td>
<td>ISO-8859-7, ISO_8859-7</td>
</tr>
<tr>
<td>819</td>
<td>ISO-8859-1, ISO_8859-1</td>
</tr>
<tr>
<td>920</td>
<td>ISO-8859-9, ISO_8859-9</td>
</tr>
<tr>
<td>1200</td>
<td>UTF-16</td>
</tr>
</tbody>
</table>
Exceptions

For most exceptions, the XML text contains the part of the document that was parsed up to and 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 either zero or the XML text contains just the encoding declaration value from the document. The example above 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, 100,001 through 165,535, or 200,001 to 265,535. The exception event in the example above has an exception number of 1 and thus is continuable.

2. Fatal exceptions, which don’t allow continuation. Fatal exceptions have exception codes greater than 99 (but less than 100,000).

Returning from the exception event function with a non-zero 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 section 2.5.6.1, "Continuable exceptions" 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 100,001 through 165,535 and 200,001 through 265,535. 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 100,000 for EBCDIC CCSIDs, or 200,000 for ASCII CCSIDs. For instance, if the exception code contains 101,140, the document’s CCSID is 01140. The CCSID value provided by the PLISAXA or PLISAXB built-in subroutine is either set 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 100,000 or 200,000), the parser proceeds using the document’s CCSID. This is the only case where the parser continues after a non-zero 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

The following example illustrates the use of the PLISAXA built-in subroutine and uses the example XML document cited above:

```
saxtest: package exports(saxtest);

  define alias event
    limited entry( pointer, pointer, fixed bin(31) )
    returns( byvalue fixed bin(31) )
    options( byvalue );

  define alias event_end_of_document
    limited entry( pointer )
    returns( byvalue fixed bin(31) )
    options( byvalue );

  define alias event_predefined_ref
    limited entry( pointer, char(1) )
    returns( byvalue fixed bin(31) )
    options( byvalue nodescription );

  define alias event_character_ref
    limited entry( pointer, fixed bin(31) )
    returns( byvalue fixed bin(31) )
    options( byvalue );

  define alias event_pi
    limited entry( pointer, pointer, fixed bin(31),
                 pointer, fixed bin(31) )
    returns( byvalue fixed bin(31) )
    options( byvalue );

  define alias event_exception
    limited entry( pointer, pointer, fixed bin(31),
                 fixed bin(31) )
    returns( byvalue fixed bin(31) )
    options( byvalue );
```

Figure 44. PLISAXA coding example - type declarations
saxtest: proc options( main );

dcl 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 45. 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's best"/>'
|| '<?spread please use real mayonnaise ?>'
|| '<meat>Ham & 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 46. PLISAXA coding example - main routine*
dcl chars char(32000) based;

start_of_document:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );
  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 );
  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 );
  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 47. PLISAXA coding example - event routines
standalone_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

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 );

dcl userToken pointer;
dcl xmlToken pointer;
dcl tokenLength fixed bin(31);

  put skip list( lowercase( procname() )
    || ' ' || substr(xmlToken->chars,1,tokenLength ) || '>' );

  return(0);
end;

eof_document:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue );

dcl userToken pointer;

  put skip list( lowercase( procname() ) );

  return(0);
end;

PLISAXA coding example - event routines (continued)
**PLISAXA coding example - event routines (continued)**

```plaintext
start_of_element:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue );

    dcl userToken    pointer;
    dcl xmlToken     pointer;
    dcl tokenLength  fixed bin(31);

    put skip list( lowercase( procname() )
        || ' ' || substr(xmlToken->chars,i,TokenLength ) || ' ' );

    return(0);
end;

attribute_name:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue );

    dcl userToken    pointer;
    dcl xmlToken     pointer;
    dcl tokenLength  fixed bin(31);

    put skip list( lowercase(procname() )
        || ' ' || substr(xmlToken->chars,i,TokenLength ) || ' ' );

    return(0);
end;

attribute_characters:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue );

    dcl userToken    pointer;
    dcl xmlToken     pointer;
    dcl tokenLength  fixed bin(31);

    put skip list( lowercase(procname() )
        || ' ' || substr(xmlToken->chars,i,TokenLength ) || ' ' );

    return(0);
end;
```
attribute_predefined_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue 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 );

  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 );

  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)
proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue );

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_ofCDATA:
proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue );

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 );

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 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);

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);

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)
PLISAXA coding example - event routines (continued)
The preceding program would produce the following output:
Continuable exception codes

For each value of the exception code parameter passed to the exception event (listed under the heading "Number"), the following table describes the exception, and 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 33. Continuable Exceptions

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Parser Action on Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The parser found an invalid character while scanning white space outside element content.</td>
<td>The parser generates a content_characters event with XML text containing the (single) invalid character. Parsing continues at the character after the invalid character.</td>
</tr>
<tr>
<td>2</td>
<td>The parser found an invalid start of a processing instruction, element, comment or document type declaration outside element content.</td>
<td>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.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
<td>Parser Action on Continuation</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>3</td>
<td>The parser found a duplicate attribute name.</td>
<td>The parser generates an attribute_name event with the XML text containing the duplicate attribute name.</td>
</tr>
<tr>
<td>4</td>
<td>The parser found the markup character &quot;&lt;&quot; in an attribute value.</td>
<td>Prior to generating the exception event, the parser generates an attribute_characters event for any part of the attribute value prior to the &quot;&lt;&quot; character. After the exception event, the parser generates an attribute_characters event with XML text containing &quot;&lt;&quot;. Parsing then continues at the character after the &quot;&lt;&quot;.</td>
</tr>
<tr>
<td>5</td>
<td>The start and end tag names of an element did not match.</td>
<td>The parser generates an end_of_element event with XML text containing the mismatched end name.</td>
</tr>
<tr>
<td>6</td>
<td>The parser found an invalid character in element content.</td>
<td>The parser includes the invalid character in XML text for the subsequent content_characters event.</td>
</tr>
<tr>
<td>7</td>
<td>The parser found an invalid start of an element, comment, processing instruction or CDATA section in element content.</td>
<td>Prior to generating the exception event, the parser generates a content_characters event for any part of the content prior to the &quot;&lt;&quot; markup character. After the exception event, the parser generates a content_characters event with XML text containing 2 characters: the &quot;&lt;&quot; followed by the invalid character. Parsing continues at the character after the invalid character.</td>
</tr>
<tr>
<td>8</td>
<td>The parser found in element content the CDATA closing character sequence &quot;]]&gt;&quot; without the matching opening character sequence &quot;&lt;![CDATA[&quot;.</td>
<td>Prior to generating the exception event, the parser generates a content_characters event for any part of the content prior to the &quot;]]&gt;&quot; character sequence. After the exception event, the parser generates a content_characters event with XML text containing the 3-character sequence &quot;]]&gt;&quot;. Parsing continues at the character after this sequence.</td>
</tr>
<tr>
<td>9</td>
<td>The parser found an invalid character in a comment.</td>
<td>The parser includes the invalid character in XML text for the subsequent comment event.</td>
</tr>
<tr>
<td>10</td>
<td>The parser found in a comment the character sequence &quot;--&quot; not followed by &quot;&gt;&quot;.</td>
<td>The parser assumes that the &quot;--&quot; character sequence terminates the comment, and generates a comment event. Parsing continues at the character after the &quot;--&quot; sequence.</td>
</tr>
<tr>
<td>11</td>
<td>The parser found an invalid character in a processing instruction data segment.</td>
<td>The parser includes the invalid character in XML text for the subsequent processing_instruction event.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
<td>Parser Action on Continuation</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>12</td>
<td>A processing instruction target name was “xml” in lower-case, upper-case or mixed-case.</td>
<td>The parser generates a processing_instruction event with XML text containing “xml” in the original case.</td>
</tr>
<tr>
<td>13</td>
<td>The parser found an invalid digit in a hexadecimal character reference (of the form &amp;#xddd).</td>
<td>The parser generates an attribute_characters or content_characters event with XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.</td>
</tr>
<tr>
<td>14</td>
<td>The parser found an invalid digit in a decimal character reference (of the form &amp;#dd).</td>
<td>The parser generates an attribute_characters or content_characters event with XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.</td>
</tr>
<tr>
<td>15</td>
<td>The encoding declaration value in the XML declaration did not begin with lower- or upper-case A through Z</td>
<td>The parser generates the encoding event with XML text containing the encoding declaration value as it was specified.</td>
</tr>
<tr>
<td>16</td>
<td>A character reference did not refer to a legal XML character.</td>
<td>The parser generates an attribute_character_reference or content_character_reference event with XML-NTEXT containing the single Unicode character specified by the character reference.</td>
</tr>
<tr>
<td>17</td>
<td>The parser found an invalid character in an entity reference name.</td>
<td>The parser includes the invalid character in the XML text for the subsequent unknown_attribute_reference or unknown_content_reference event.</td>
</tr>
<tr>
<td>18</td>
<td>The parser found an invalid character in an attribute value.</td>
<td>The parser includes the invalid character in XML text for the subsequent attribute_characters event.</td>
</tr>
<tr>
<td>50</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>51</td>
<td>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.</td>
<td>The parser uses the encoding specified by the document encoding declaration.</td>
</tr>
<tr>
<td>52</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
<td>Parser Action on Continuation</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>53</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>54</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>55</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>56</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>57</td>
<td>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.</td>
<td>The parser uses the encoding specified by the document encoding declaration.</td>
</tr>
<tr>
<td>58</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>59</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>60</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
</tbody>
</table>
Table 33. Continuable Exceptions (continued)

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Parser Action on Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>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.</td>
<td>The parser uses the encoding specified by the CODEPAGE compiler option.</td>
</tr>
<tr>
<td>100,001 through 165,535</td>
<td>The document was encoded in EBCDIC, and the encodings specified by the CODEPAGE compiler option and 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 100,000.</td>
<td>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 100,000 from the exception code), the parser uses this encoding.</td>
</tr>
<tr>
<td>200,001 through 265,535</td>
<td>The document was encoded in ASCII, and the encodings specified by the CODEPAGE compiler option and 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 200,000.</td>
<td>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 200,000 from the exception code), the parser uses this encoding.</td>
</tr>
</tbody>
</table>

Terminating exception codes

Table 34. Terminating Exceptions

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>The parser reached the end of the document while scanning the start of the XML declaration.</td>
</tr>
<tr>
<td>101</td>
<td>The parser reached the end of the document while looking for the end of the XML declaration.</td>
</tr>
<tr>
<td>102</td>
<td>The parser reached the end of the document while looking for the root element.</td>
</tr>
<tr>
<td>103</td>
<td>The parser reached the end of the document while looking for the version information in the XML declaration.</td>
</tr>
<tr>
<td>104</td>
<td>The parser reached the end of the document while looking for the version information value in the XML declaration.</td>
</tr>
<tr>
<td>106</td>
<td>The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.</td>
</tr>
<tr>
<td>108</td>
<td>The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.</td>
</tr>
<tr>
<td>109</td>
<td>The parser reached the end of the document while scanning an attribute name.</td>
</tr>
<tr>
<td>110</td>
<td>The parser reached the end of the document while scanning an attribute value.</td>
</tr>
<tr>
<td>111</td>
<td>The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.</td>
</tr>
<tr>
<td>112</td>
<td>The parser reached the end of the document while scanning an empty element tag.</td>
</tr>
<tr>
<td>113</td>
<td>The parser reached the end of the document while scanning the root element name.</td>
</tr>
<tr>
<td>114</td>
<td>The parser reached the end of the document while scanning an element name.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>115</td>
<td>The parser reached the end of the document while scanning character data in element content.</td>
</tr>
<tr>
<td>116</td>
<td>The parser reached the end of the document while scanning a processing instruction in element content.</td>
</tr>
<tr>
<td>117</td>
<td>The parser reached the end of the document while scanning a comment or CDATA section in element content.</td>
</tr>
<tr>
<td>118</td>
<td>The parser reached the end of the document while scanning a comment in element content.</td>
</tr>
<tr>
<td>119</td>
<td>The parser reached the end of the document while scanning a CDATA section in element content.</td>
</tr>
<tr>
<td>120</td>
<td>The parser reached the end of the document while scanning a character reference or entity reference in element content.</td>
</tr>
<tr>
<td>121</td>
<td>The parser reached the end of the document while scanning after the close of the root element.</td>
</tr>
<tr>
<td>122</td>
<td>The parser found a possible invalid start of a document type declaration.</td>
</tr>
<tr>
<td>123</td>
<td>The parser found a second document type declaration.</td>
</tr>
<tr>
<td>124</td>
<td>The first character of the root element name was not a letter, '_' or ':'</td>
</tr>
<tr>
<td>125</td>
<td>The first character of the first attribute name of an element was not a letter, '_' or ':'</td>
</tr>
<tr>
<td>126</td>
<td>The parser found an invalid character either in or following an element name.</td>
</tr>
<tr>
<td>127</td>
<td>The parser found a character other than '=' following an attribute name.</td>
</tr>
<tr>
<td>128</td>
<td>The parser found an invalid attribute value delimiter.</td>
</tr>
<tr>
<td>130</td>
<td>The first character of an attribute name was not a letter, '_' or ':'</td>
</tr>
<tr>
<td>131</td>
<td>The parser found an invalid character either in or following an attribute name.</td>
</tr>
<tr>
<td>132</td>
<td>An empty element tag was not terminated by a ']' following the '/'.</td>
</tr>
<tr>
<td>133</td>
<td>The first character of an element end tag name was not a letter, '_' or ':'</td>
</tr>
<tr>
<td>134</td>
<td>An element end tag name was not terminated by a ']'</td>
</tr>
<tr>
<td>135</td>
<td>The first character of an element name was not a letter, '_' or ':'</td>
</tr>
<tr>
<td>136</td>
<td>The parser found an invalid start of a comment or CDATA section in element content.</td>
</tr>
<tr>
<td>137</td>
<td>The parser found an invalid start of a comment.</td>
</tr>
<tr>
<td>138</td>
<td>The first character of a processing instruction target name was not a letter, '_' or ':'</td>
</tr>
<tr>
<td>139</td>
<td>The parser found an invalid character in or following a processing instruction target name.</td>
</tr>
<tr>
<td>140</td>
<td>A processing instruction was not terminated by the closing character sequence '?&gt;}'.</td>
</tr>
<tr>
<td>141</td>
<td>The parser found an invalid character following '&amp;' in a character reference or entity reference.</td>
</tr>
<tr>
<td>142</td>
<td>The version information was not present in the XML declaration.</td>
</tr>
<tr>
<td>143</td>
<td>'version' in the XML declaration was not followed by a '='.</td>
</tr>
<tr>
<td>144</td>
<td>The version declaration value in the XML declaration is either missing or improperly delimited.</td>
</tr>
<tr>
<td>145</td>
<td>The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.</td>
</tr>
<tr>
<td>146</td>
<td>The parser found an invalid character following the version information value closing delimiter in the XML declaration.</td>
</tr>
<tr>
<td>147</td>
<td>The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>148</td>
<td>'encoding' in the XML declaration was not followed by a '='.</td>
</tr>
<tr>
<td>149</td>
<td>The encoding declaration value in the XML declaration is either missing or improperly delimited.</td>
</tr>
<tr>
<td>150</td>
<td>The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.</td>
</tr>
<tr>
<td>151</td>
<td>The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.</td>
</tr>
<tr>
<td>152</td>
<td>The parser found an invalid attribute instead of the optional standalone declaration in the XML declaration.</td>
</tr>
<tr>
<td>153</td>
<td>'standalone' in the XML declaration was not followed by a '='.</td>
</tr>
<tr>
<td>154</td>
<td>The standalone declaration value in the XML declaration is either missing or improperly delimited.</td>
</tr>
<tr>
<td>155</td>
<td>The standalone declaration value was neither 'yes' nor 'no' only.</td>
</tr>
<tr>
<td>156</td>
<td>The standalone declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.</td>
</tr>
<tr>
<td>157</td>
<td>The parser found an invalid character following the standalone declaration value closing delimiter in the XML declaration.</td>
</tr>
<tr>
<td>158</td>
<td>The XML declaration was not terminated by the proper character sequence '?&gt;', or contained an invalid attribute.</td>
</tr>
<tr>
<td>159</td>
<td>The parser found the start of a document type declaration after the end of the root element.</td>
</tr>
<tr>
<td>160</td>
<td>The parser found the start of an element after the end of the root element.</td>
</tr>
<tr>
<td>300</td>
<td>The document was encoded in EBCDIC, but the CODEPAGE compiler option specified a supported ASCII code page.</td>
</tr>
<tr>
<td>301</td>
<td>The document was encoded in EBCDIC, but the CODEPAGE compiler option specified Unicode.</td>
</tr>
<tr>
<td>302</td>
<td>The document was encoded in EBCDIC, but the CODEPAGE compiler option specified an unsupported code page.</td>
</tr>
<tr>
<td>303</td>
<td>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.</td>
</tr>
<tr>
<td>304</td>
<td>The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document did not contain an encoding declaration.</td>
</tr>
<tr>
<td>305</td>
<td>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.</td>
</tr>
<tr>
<td>306</td>
<td>The document was encoded in ASCII, but the CODEPAGE compiler option specified a supported EBCDIC code page.</td>
</tr>
<tr>
<td>307</td>
<td>The document was encoded in ASCII, but the CODEPAGE compiler option specified Unicode.</td>
</tr>
<tr>
<td>308</td>
<td>The document was encoded in ASCII, but the CODEPAGE compiler option did not specify a supported EBCDIC code page, ASCII or Unicode.</td>
</tr>
<tr>
<td>309</td>
<td>The CODEPAGE compiler option specified a supported ASCII code page, but the document was encoded in Unicode.</td>
</tr>
<tr>
<td>310</td>
<td>The CODEPAGE compiler option specified a supported EBCDIC code page, but the document was encoded in Unicode.</td>
</tr>
<tr>
<td>311</td>
<td>The CODEPAGE compiler option specified an unsupported code page, but the document was encoded in Unicode.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>312</td>
<td>The document was encoded in ASCII, but both the encodings provided externally and within the document encoding declaration are unsupported.</td>
</tr>
<tr>
<td>313</td>
<td>The document was encoded in ASCII, but the CODEPAGE compiler option is unsupported and the document did not contain an encoding declaration.</td>
</tr>
<tr>
<td>314</td>
<td>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.</td>
</tr>
<tr>
<td>315</td>
<td>The document was encoded in UTF-16 Little Endian, which the parser does not support on this platform.</td>
</tr>
<tr>
<td>316</td>
<td>The document was encoded in UCS4, which the parser does not support.</td>
</tr>
<tr>
<td>317</td>
<td>The parser cannot determine the document encoding. The document may be damaged.</td>
</tr>
<tr>
<td>318</td>
<td>The document was encoded in UTF-8, which the parser does not support.</td>
</tr>
<tr>
<td>319</td>
<td>The document was encoded in UTF-16 Big Endian, which the parser does not support on this platform.</td>
</tr>
<tr>
<td>500 to 99,999</td>
<td>Internal error. Please report the error to your service representative.</td>
</tr>
</tbody>
</table>
Chapter 26. Using PL/I MLE in your applications

With the introduction of MLE, PL/I for Windows allows support for a number of additional language features. The purpose of this chapter is for you to become familiar with the new attribute, compile-time options, date patterns, and built-in functions. As you follow the sequence of the chapter, you should have an idea about how to apply these to your existing applications.

Applying attributes and options

The language features introduced in these sections are found elsewhere in the Programming Guide and Language Reference, but are repeated in this chapter so you can better understand how they work together.

DATE attribute

Implicit date comparisons and conversions are made by the compiler if the two operands have the DATE attribute. The DATE attribute specifies that a variable, argument, or returned value holds a date with a specified pattern. millennium language extensions supports a number of date patterns as described in "Understanding date patterns" on page 483.

pattern
One of the supported date patterns. If you do not specify a pattern, YYMMDD is the default.

The DATE attribute is valid only with variables having one of the following sets of attributes:
- CHAR(n) NONVARYING
- PIC(n)9' REAL
- FIXED DEC(n,0) REAL

The length or precision, n, must be a constant equal to the length of the date pattern or default pattern.

When the RESPECT compile-time option (discussed later in this chapter) has been specified, the DATE built-in function returns a value that has the attribute DATE('YYMMDD'). This allows DATE() to be assigned to a variable with the attribute DATE('YYMMDD') without an error message being generated. If DATE() is assigned to a variable not having the DATE attribute, however, an error message is generated.

Here are a few examples using the DATE attribute:

dcl gregorian_Date char(6) date;

dcl julian_Date pic'(5)9' date ('YYDDD');

dcl year fixed dec(2) date('YY');

The DATE attribute is useful even if you have no year 2000 problems in your applications. You can use it to manipulate differing dates as shown in these examples:
**DATE attribute**

dcl gregorian_Date char(8) date ('YYYYMMDD');
dcl julian_Date pic'(7)9' date ('YYYYDDD');
if julian_Date > gregorian_Date then ...

**RESPECT compile-time option**

Use the RESPECT option to specify which attributes the compiler should recognize. Currently, DATE is the only selection possible for this compile-time option.

The default is RESPECT() and causes the compiler to ignore any specification of the DATE attribute. Therefore, the DATE attribute is not applied to the result of DATE built-in. NORESPECT is a synonym for RESPECT().

Specifying RESPECT(DATE), on the other hand, causes the compiler to honor any specification of the DATE attribute and to apply the DATE attribute to the result of DATE built-in.

RESPECT() is not accepted when compiling with the PLI command on TSO/MVS.

**WINDOW compile-time option**

By default, all dates with two-digit years are viewed as falling in a window starting with 1950 and ending in 2049. You can use the WINDOW option to change the value for your century window.

As previously mentioned, the default for this option is WINDOW(1950). You can specify the value for w as one of the following:
- An unsigned integer between 1582 and 9999 (inclusive) that represents the start of a fixed century window
- A negative integer between -1 and -99 (inclusive) that creates a "sliding" century window
- Zero, indicating the value for w is the current year.

To create a fixed window, you could specify WINDOW(1900) and all two-digit years would be assumed to occur in the 20th century.

If the current year were 1998, and you wanted to create a sliding window, you could specify WINDOW(-5). The resulting century window would span the years 1993 through 2092, inclusive. When the year changes to 1999, the window would also move forward by one year.

If you set a value for the century window using the WINDOW compile-time option, that value is used for the window argument in the built-in functions which allow it, unless otherwise specified in that built-in. See “Using built-in functions with MLE” on page 484 for more details.

**RULES compile-time option**

In general, the RULES option allows or disallows certain language capabilities and allows you to choose semantics when alternatives are available. Currently, LAXCOMMENT is the only selection available for this option.

The default is RULES(NOLAXCOMMENT). LAXCOM and NOLAXCOM are acceptable abbreviations for the suboptions.
RULES compile-time option

If you specify RULES(LAXCOMMENT), the compiler ignores the special characters /*/; therefore, whatever comes between the sets of characters is interpreted as part of the syntax instead of as a comment. If you specify RULES(NOLAXCOMMENT), the compiler treats /*/ as the start of a comment which continues until a closing */ is found.

If you happen to have workstation code that you are porting to the mainframe and uses /*/ around the DATE attribute, you need to use the RULES(LAXCOMMENT) option so that the compiler honors the attribute.

Understanding date patterns

PL/I MLE supports a series of date patterns as shown in the following table.

**Table 35. Date patterns supported by PL/I MLE**

<table>
<thead>
<tr>
<th>Pattern Type</th>
<th>4-digit year Example</th>
<th>2-digit year Example</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year first</td>
<td>YYYY</td>
<td>YY</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>1999</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td></td>
<td>YYYYY</td>
<td>YYMM</td>
<td>9912</td>
</tr>
<tr>
<td></td>
<td>199912</td>
<td>YYMMDD</td>
<td>991225</td>
</tr>
<tr>
<td></td>
<td>YYYYMM</td>
<td>YYMM</td>
<td>99DEC</td>
</tr>
<tr>
<td></td>
<td>1999DEC</td>
<td>YYMMDD</td>
<td>99DEC25</td>
</tr>
<tr>
<td></td>
<td>YYYYMMDD</td>
<td>YYMMDD</td>
<td>99DEC25</td>
</tr>
<tr>
<td></td>
<td>YYMMmm</td>
<td>YYMM</td>
<td>99Dec</td>
</tr>
<tr>
<td></td>
<td>1999Dec</td>
<td>YYmm</td>
<td>99Dec</td>
</tr>
<tr>
<td></td>
<td>YYYYmmmDD</td>
<td>YYmmDD</td>
<td>99Dec25</td>
</tr>
<tr>
<td></td>
<td>1999Dec25</td>
<td>YYmmDD</td>
<td>99Dec25</td>
</tr>
<tr>
<td></td>
<td>YYYYDDDD</td>
<td>YYDDDD</td>
<td>99359</td>
</tr>
<tr>
<td></td>
<td>1999359</td>
<td>YYDDDD</td>
<td>99359</td>
</tr>
<tr>
<td>Month first</td>
<td>MMMYYYY</td>
<td>MMYY</td>
<td>1299</td>
</tr>
<tr>
<td></td>
<td>121999</td>
<td>MMYY</td>
<td>1299</td>
</tr>
<tr>
<td></td>
<td>MMDDYYYY</td>
<td>MMDDYY</td>
<td>122599</td>
</tr>
<tr>
<td></td>
<td>12251999</td>
<td>MMDDYY</td>
<td>122599</td>
</tr>
<tr>
<td></td>
<td>MMMYYY</td>
<td>MMMY</td>
<td>DEC99</td>
</tr>
<tr>
<td></td>
<td>DEC1999</td>
<td>MMMY</td>
<td>DEC99</td>
</tr>
<tr>
<td></td>
<td>MMMDDYYYY</td>
<td>MMMDDYY</td>
<td>DEC2599</td>
</tr>
<tr>
<td></td>
<td>DEC251999</td>
<td>MMMDDYY</td>
<td>DEC2599</td>
</tr>
<tr>
<td></td>
<td>MmmYYYY</td>
<td>MmmY</td>
<td>Dec99</td>
</tr>
<tr>
<td></td>
<td>Dec1999</td>
<td>MmmY</td>
<td>Dec99</td>
</tr>
<tr>
<td></td>
<td>MmmDDYYYY</td>
<td>MmmDDYY</td>
<td>Dec2599</td>
</tr>
<tr>
<td></td>
<td>Dec251999</td>
<td>MmmDDYY</td>
<td>Dec2599</td>
</tr>
<tr>
<td>Day first</td>
<td>DDMMYYYYY</td>
<td>DDMMYY</td>
<td>251299</td>
</tr>
<tr>
<td></td>
<td>25121999</td>
<td>DDMMYY</td>
<td>251299</td>
</tr>
<tr>
<td></td>
<td>DDMMMYYYY</td>
<td>DDMMMYY</td>
<td>25DEC99</td>
</tr>
<tr>
<td></td>
<td>25DEC1999</td>
<td>DDMMMYY</td>
<td>25DEC99</td>
</tr>
<tr>
<td></td>
<td>DDMmmYYYY</td>
<td>DDMmmYY</td>
<td>25Dec99</td>
</tr>
<tr>
<td></td>
<td>25Dec1999</td>
<td>DDMmmYY</td>
<td>25Dec99</td>
</tr>
<tr>
<td></td>
<td>DDDYYYY</td>
<td>DDDYY</td>
<td>35999</td>
</tr>
<tr>
<td></td>
<td>3591999</td>
<td>DDDYY</td>
<td>35999</td>
</tr>
</tbody>
</table>

When the day or month is omitted from one of these patterns, the compiler assumes it has a value of 1.

If the day or month are not omitted but out of range, for example 00/38/11, a message is issued if the date involves a comparison. Exceptions to the rules are cases of patterns YYMM and YYMMDD with values of all zeros that will be converted to a Julian date of 1, that is, the smallest valid date.

Patterns and windowing

To define how a date with a two-digit year (YY) is interpreted, a century window is defined using the WINDOW compile-time option. As described previously, the century window defines the beginning of a 100-year span to which the two-digit year applies.

Without the help of PL/I's Millennium Language Extensions, you would have to implement something like the following logic which converts y2 from a two-digit year to a four-digit year with a window (w).

dcl y4 pic'9999';
dcl cc pic'99';
Patterns and windowing

\[
cc = w/100;
\]
\[
\text{if } y2 \lt \text{mod}(w,100) \text{ then}
\]
\[
y4 = (100 \times cc) + 100 + y2;
\]
\[
\text{else}
\]
\[
y4 = (100 \times cc) + y2;
\]

Using this example, if you were to specify \texttt{WINDOW(1900)}, 19 would be interpreted as the year 1919. If you were to specify \texttt{WINDOW(1950)}, however, 19 would be interpreted as the year 2019.

Conversely, this logic calculates the two-digit year (\(y2\)) when converting from a four-digit year.

\[
dcl y4 \text{ pic}'9999';
\]
\[
\text{if } y4 < w | y4 >= w + 100 \text{ then}
\]
\[
signal \text{ error;}
\]
\[
y2 = \text{mod}(y4,100);
\]

Using built-in functions with MLE

The date patterns for PL/I MLE are supported by the \texttt{DAYS} and \texttt{DAYSTODATE} built-in functions. These built-ins both accept the optional argument (\(w\)) that specifies a window to be used in handling two-digit year patterns. If you specify \(w\) as part of \texttt{DAYS} or \texttt{DAYSTODATE}, the value you enter overrides the value as defined by the \texttt{WINDOW} compile-time option.

**DAYS**

\texttt{DAYS} returns a \texttt{FIXED BINARY(31,0)} value which is the number of days (in Lilian format) corresponding to the date \(d\).

\(d\) String expression representing a date. If omitted, it is assumed to be the value returned by \texttt{DATETIME()}.

The value for \(d\) should have character type. If not, \(d\) is converted to character.

\(p\) One of the supported date patterns shown in Table 35 on page 483. If omitted, the compiler assumes that \(p\) is the default pattern returned by the \texttt{DATETIME} built-in function (YYYYMMDDHHMISS999).

\(p\) should have character type. If not, it is converted to character.

\(w\) An integer expression that defines a century window to be used to handle any two-digit year formats.

- If the value is positive, such as 1950, it is treated as a year.
- If negative or zero, the value specifies an offset to be subtracted from the current, system-supplied year.
- If omitted, \(w\) defaults to the value specified in the \texttt{WINDOW} compile-time option.

The following example shows uses of both the \texttt{DAYS} and \texttt{DAYSTODATE} built-in functions:

\[
dcl date_format char(8) static init('MMDYYYY');
\]
\[
dcl todays_date char(8);
\]
\[
dcl sep2_1993 char(8);
\]
\[
dcl days_of_july4_1993 fixed bin(31);
\]
\[
dcl msg char(100) varying;
\]
\[
dcl date_due char(8);
\]
todays_date = daystodate(days(), date_format);

days_of_july4_1993 = days('07041993', 'MMDDYYYY');
sep2_1993 = daystodate(days_of_july4_1993 + 60, Date_format);
/* 09021993 */

date_due = daystodate(days() + 60, date_format);
/* assuming today is July 4, 1993, this would be Sept. 2, 1993 */

msg = 'Please pay amount due on or before ' ||
     substr(date_due, 1, 2) || '/' ||
     substr(date_due, 3, 2) || '/' ||
     substr(date_due, 5);

---

**DAYSTODATE**

DAYSTODATE returns a nonvarying character string containing the date in the form p that corresponds to d days (in Lilian format).

- **d** The number of days (in Lilian format).
  - d must have a computational type and is converted to FIXED BINARY(31,0) if necessary.

- **p** One of the supported date patterns shown in Table 35 on page 483. If omitted, the compiler assumes that p is the default pattern returned by the DATETIME built-in function (YYYYMMDDHHMISS999).
  - p should have character type. If not, it is converted to character.

- **w** An integer expression that defines a century window to be used to handle any two-digit year formats.
  - If the value is positive, such as 1950, it is treated as a year.
  - If negative or zero, the value specifies an offset to be subtracted from the current, system-supplied year.
  - If omitted, w defaults to the value specified in the WINDOW compile-time option.

---

**Performing date calculations and comparisons**

Once you understand what the PL/I millennium language features are and you have made the appropriate syntax changes, you can use MLE to perform calculations and comparisons in your applications.

**Explicit date calculations**

You can use the DAYS and DAYSTODATE built-in functions to make date comparisons and calculations manually.

**Comparing dates**

To compare two dates d1 and d2 which have the date pattern YYMMDD, you can use the following code:

```
DAYS (d1, 'YYMMDD', w) < DAYS(d2, 'YYMMDD', w)
```

**Converting dates**

You can convert between a two-digit date (d1) with the pattern YYMMDD and a four-digit date (d2) with the pattern YYYYMMDD using assignments:

```
d2 = daystodate(days(d1, 'YYMMDD', w), 'YYYYMMDD');
d1 = daystodate(days(d2, 'YYYYMMDD'), 'YYMMDD', w);
```
Implicit date calculations

You can use MLE to take advantage of implicit date comparisons and conversions if you first complete the following steps:

- Give the two operands the DATE attribute
- Specify the RESPECT compile-time option

Implicit date comparisons

The DATE attribute causes implicit commoning when two variables declared with the DATE attribute are compared. Comparisons where only one variable has the DATE attribute are flagged, and the other comparand is generally treated as if it had the same DATE attribute, although some exceptions apply which are discussed later.

Implicit commoning means that the compiler generates code to convert the dates to a common, comparable representation. This process converts 2-digit years using the window you specify in the WINDOW compile-time option.

In the following code fragment, if the DATE attribute is honored, then the comparison in the second display statement is 'windowed'. This means that if the window started at 1900, the comparison would return false. However, if the window started at 1950, the comparison would return true.

```pli
DCL a PIC'(6)9' DATE;
DCL b PIC'(6)9' def(a);
DCL c PIC'(6)9' DATE;
DCL d PIC'(6)9' def(c);

B = '670101';
D = '010101';

DISPLAY( B || '<' || D || '?' );
DISPLAY( A < C );
```

Date comparisons can also occur in the following places:

- IF and SELECT statements
- WHILE or UNTIL clauses
- Implicit comparisons caused by a TO clause.

Comparing dates with like patterns

The compiler does not generate any special code to compare dates with identical patterns under the following conditions:

- The comparison operator of = or ~= is used
- The pattern is equal to YYYY, YYYYMM, YYYYDDD, or YYYYMMDD.

Comparing dates with differing patterns

For comparisons involving dates with unlike patterns, the compiler generates code to convert the dates to a common comparable representation. Once the conversion has taken place, the compiler compares the two values.
Implicit date comparisons

Comparisons involving the DATE attribute and a literal
If you are making comparisons in which one comparand has the DATE attribute and the other is a literal, the compiler issues a W-level message. Further compiler action depends on the value of the literal as follows:

- If the literal appears to be a valid date, it is treated as if it had the same date pattern and window as the comparand with the DATE attribute.
- If the literal does not appear to be a valid date, the DATE attribute is ignored on the other comparand.

```pli
dcl start_date char(6) date;
if start_date >= '' then /* no windowing */
...
if start_date >= '851003' then /* windowed */
...
```

Comparisons involving the DATE attribute and a non-literal
In comparisons where one comparand has the DATE attribute and the other is not a date and not a literal, the compiler issues an E-level message. The non-date value is treated as if it had the same date pattern as the other comparand and as if it had the same window.

```pli
dcl start_date char(6) date;
dcl non_date char (6);
if start_date >= non_date then /* windowed */
...
```

Implicit DATE assignments
The DATE attribute can also cause implicit conversions to occur in assignments of two variables declared with date patterns.

- If the source and target have the same DATE and data attributes, then the assignment proceeds as if neither had the DATE attribute.
- If the source and target have differing DATE attributes, then the compiler generates code to convert the source date before making the assignment.
- In assignments where the source has the DATE attribute but the target does not, the compiler issues an E-level message and ignores the DATE attribute.
- In assignments where the target has the DATE attribute but the source does not (and the source IS NOT a literal), the compiler issues an E-level message and ignores the DATE attribute.
- In assignments where the target has the DATE attribute but the source does not (and the source IS a literal), the compiler issues a W-level message and ignores the DATE attribute.

```pli
dcl start_date char(6) date;
start_date = '';
...
```

- If the source holds a four-digit year and the target holds a two-digit year, the source can hold a year that is not in the target window. In this case, the ERROR condition is raised.

```pli
dcl x char(6) date;
dcl y char(8) date('YYYYMMDD');
y = '20600101';
x = y; /* raises error if window is <= 1960 */
```

- The DATE attribute is ignored in:
  - The debugger
  - Assignments performed in record I/O statements
Implicit DATE assignments

- Assignments and conversions performed in stream I/O statements (such as GET DATA).

Even if you do not choose a windowing solution, you might have some code that needs to manipulate both two- and four-digit years. You can use multiple date patterns to help you in these situations:

```pli
 dcl old_date char(6) date('YYMMDD');
dcl new_date char(8) date('YYYYMMDD');

new_date = old_date;
```
Part 7. Appendixes
Notices

Programming interface information . . . . . . . 492
Macros for customer use . . . . . . . . . . . . . 492
Trademarks . . . . . . . . . . . . . . . . . . . . 492

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74/G4
Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM PL/I for Windows.

Macros for customer use

IBM PL/I for Windows provides no macros that allow a customer installation to write programs that use the services of IBM PL/I for Windows.

Attention: Do not use as programming interfaces any IBM PL/I for Windows macros.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

<table>
<thead>
<tr>
<th>Term</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>IMS/ESA</td>
</tr>
<tr>
<td>CICS</td>
<td>Language</td>
</tr>
<tr>
<td>CICS/ESA</td>
<td>Environment</td>
</tr>
<tr>
<td>DFSMS/MVS</td>
<td>OS/2</td>
</tr>
<tr>
<td>DFSORT</td>
<td>OS/390</td>
</tr>
<tr>
<td>IBM</td>
<td>Proprioter</td>
</tr>
<tr>
<td>IMS</td>
<td>Rational</td>
</tr>
<tr>
<td></td>
<td>VisualAge</td>
</tr>
<tr>
<td></td>
<td>WebSphere</td>
</tr>
</tbody>
</table>

Windows is a trademark of Microsoft Corporation in the United States and/or other countries.
UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.
Bibliography

PL/I publications .......................................... 495  Related publications ......................................... 495
PL/I for AIX ................................................. 495  DB2 UDB for OS/390 and z/OS .................................. 495
Enterprise PL/I publications ................................. 495  TXSeries for Multiplatforms .................................. 495

PL/I publications

PL/I for AIX
Programming Guide, SC14-7319
Language Reference, SC14-7320
Messages and Codes, GC14-7321
Installation Guide, GC14-7322

Enterprise PL/I publications
Programming Guide, GI11-9145
Language Reference, SC14-7285
Messages and Codes, GC14-7286
Diagnosis Guide, GC27-1459
Compiler and Run-Time Migration Guide, GC14-7284

Related publications

DB2 UDB for OS/390 and z/OS
Administration Guide, SC26-9931
Command Reference, SC26-9934
SQL Reference, SC26-9944
Application Programming and SQL Guide, SC26-9933
Messages and Codes, GC26-9940

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/infcert/txformp/v7r1/index.jsp
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).

alphabetical 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.
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.

group 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

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.
character string constant
A sequence of characters enclosed in single quotes; for example, 'Shakespeare's 'Hamlet''.

character set
A defined collection of characters. See language character set and data character set. 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 listing 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 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
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.

**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**
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.

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.

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**
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, assignment 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.
**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.

**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**

? option for ILIB 391

*PROCESS statement 18

%INCLUDE statement 19

%LINE directive 19

%PROCESS statement specifying compile-time options with 25

A

access methods

DDM 229

I/O 231

accessing data sets

examples of REGIONAL(1) 272

record I/O 271

REGIONAL(1) 282

stream I/O 259

adapting existing programs for workstation VSAM 294

ADDBUFF ENVIRONMENT option 4

ADDEXT compiler option 31

adding or replacing objects in a library, ILIB 388

aggregate

length table, example 181

AGGREGATE compiler option 32

ALIGNED compile-time suboption 46

American National Standard Institute (ANSI)
in CTLASA option 234

AMTHD option 239

ANS

compile-time suboption 46

print control characters 256

ANSI

code inspection 211

code conversion tables 376

ASCII

compile-time suboption
description 46
effect on performance 360
data declarations 323

AREA's and INITIAL attribute 11

array expressions restrictions 5

ASA option 240

ASCII

compile-time suboption
description 46
effect on performance 360
data conversion tables 376

DBCS portability 10

PORTABILITY considerations 8

ASCII ENVIRONMENT option 4

ASSIGNABLE compile-time suboption 46

attributes and cross-reference table 180

ATTRIBUTES compiler option 33

AUTOPAGE option 241

avoiding calls to library routines 366

command line, setting run-time options 207

communications area, SQL 158

comparing dates

implicit 486

using literals 487

using non-literals 487

with differing patterns 486

with like patterns 485, 486

compatibility of PL/I for z/OS files for the workstation 294

compilation

compile-time options 27

environment variables

IBM.DECK 23

IBM.OBJECT 23

IBM.OPTIONS 21

IBM.PRINT 23

IBM.SOURCE 23

IBM.SYSLIB 23

INCLUDE 24

TMP 24

failure 225

mainframe applications on your workstation 3

preparing your source program 18

user exit

activating 370

customizing 371

IBMUEXIT 370

procedures 369

using the PLI command to invoke the compiler 24

compilation output

compiler output 183

using the compiler listing 175

COMPILER compiler option 40

compile-time options 4

use in debugging 213

where to specify 24

compiler

descriptions of options 27

listing

stack storage used 132

compiler options

abbreviations 27

ADDEXT 31

AGGREGATE 32

ATTRIBUTES 33

BIFPREC 34

BLANK 35

CASERULES 36

CHECK 37

CMPAT 38

CODEPAGE 39

COMPILE 40

COPYRIGHT 41

CURRENCY 42

DBCS 43

default 27

default 44, 357

calculations using dates 485

call interface conventions

with ODBC 319

calling conventions 395

general-purpose register implications

effects on performance 385

examples of passing parameters 397

parameters 397

carriage return-line feed (CR-LF) 246

CASERULES compiler option 36

CEE.OPTIONS environment variable 207

character device 230

CHECK compiler option 37

CICS

environment variables 172

IBM.PPCICS 22

preprocessor options 172

run-time user exit 375

support 170

CMPAT compiler option 38

code inspection 211

CODEPAGE compiler option 39

coding

CICS statements 173

embedded control characters 3

improving performance 361

SQL statements 158

command line parameters for ILIB 383

command line, setting run-time options 207

communications area, SQL 158

comparing dates

implicit 486

using literals 487

using non-literals 487

with differing patterns 486

with like patterns 485, 486

compatibility of PL/I for z/OS files for the workstation 294

compilation

compile-time options 27

environment variables

IBM.DECK 23

IBM.OBJECT 23

IBM.OPTIONS 21

IBM.PRINT 23

IBM.SOURCE 23

IBM.SYSLIB 23

INCLUDE 24

TMP 24

failure 225

mainframe applications on your workstation 3

preparing your source program 18

user exit

activating 370

customizing 371

IBMUEXIT 370

procedures 369

using the PLI command to invoke the compiler 24

compilation output

compiler output 183

using the compiler listing 175

COMPILER compiler option 40

compile-time options 4

use in debugging 213

where to specify 24

compiler

descriptions of options 27

listing

stack storage used 132

compiler options

abbreviations 27

ADDEXT 31

AGGREGATE 32

ATTRIBUTES 33

BIFPREC 34

BLANK 35

CASERULES 36

CHECK 37

CMPAT 38

CODEPAGE 39

COMPILE 40

COPYRIGHT 41

CURRENCY 42

DBCS 43

default 27

default 44, 357
compiler options (continued)
DEPRECATE 53
DEPRECATENEXT 55
DLLINIT 56
EXIT 57
EXTRN 58
FLAG 59
FLOATINMATH 60
GONUMBER 61, 356
GRAPHIC 62
IGNORE 63
IMPRECISE 64, 355
INCAFTER 65
INCLUDE 66
INITAUTO 67
INITBASED 68
INITCTL 69
INITSTATIC 70
LANGLVL 72
LIBS 73
LIMITS 74
LINECOUNT 75
LINEDIR 76
LIST 77
LISTVIEW 78
MACRO 80
MARGINI 81
MARGINS 82
MAXGEN 88
MAXMSG 84
MAXNEST 85
MAXSTMT 86
MAXTEMP 87
MDECK 88
MSG 89
MSGSUMMARY 90
NAMES 91
NATLANG 92
NATLANGE 92
NEST 93
NOT 94
NUMBER 95
OBJECT 96
OFFSET 97
ONSNAP 98
OPTIMIZE 99, 355
OPTIONS 100
OR 101
PP 102
PPCICS 103
PPINCLUD 104
PPLIST 105
PPMACRO 106
PPSQL 107
PPTRACE 108
PRECTYPE 109
PREFIX 357
PREFIXE 110
PROBE 111
PROCEED 112
QUOTE 114
REDUCE 115
RESEXP 116
RESPECT 117, 482
RULES 118, 356, 482
SEMANTIC 126
SNAP 127, 356
compiler options (continued)
SOSI 128
SOURCE 129
STATIC 130
STMT 131
STORAGE 132
SYNTAX 133
SYSPARM 134
SYSTEM 135
TERMINAL 136
TEST 137
USAGE 138
WIDECHAR 139
WINDOW 140, 482
XML 144
compiler restrictions
array expressions 5
built-in functions 5
DBCS 6
default statement 5
definitions of automatic variables 5
default defining 6
MACRO preprocessor 6
pseudovariables 5
RECORD I/O 4
STREAM I/O 4
structure expressions 5
concatenation 20
condition handling
coding ON-units 220
general concepts 219
interrupts 219
list of conditions and their attributes 222
qualified and unqualified conditions 222
scope and descendancy 219
terminology 219
conditions
handling conversions inline 366
handling string built-in functions inline 367
CONNECT TO statement 167
CONNECTED compile-time suboption description 47
effect on performance 359
CONSECUTIVE
files 293
option
definition 233
stream I/O 257
consecutive data sets
controlling input from the terminal capital and lowercase letters 267
end of file 267
format of data 266
stream and record files 267
using files conversationally 266
controlling output to the terminal example of an interactive program 268
format of PRINT files 268
stream and record files 268
description 255
examples 272
PRINT files 268
consecutive data sets (continued)
printer-destined files 255
using record-oriented I/O
accessing and updating a data set 271
creating a data set 271
defining files 270
ENVIRONMENT options for data transmission 271
using stream-oriented data transmission
accessing a data set with stream I/O 259
creating a data set with stream I/O 257
defining files using stream I/O 256
ENVIRONMENT options for 257
using PRINT files 261
using SYSIN and SYSPRINT files 265
control blocks
function-specific 370
global control 372
control characters
ANSI in CTLASA option 234
printer 255
conversion tables 376
converting dates 485
COPYRIGHT compiler option 41
cross-reference table in compilation output 180
CTLASA option 234
CURRENCY compile time option portability 3
CURRENCY compiler option 42
customizing
setting compile-time environment variables 21
user exit
modifying IBMUEXIT.INF 371
structure of global control blocks 372
writing your own compiler exit 372
D
data
conversion 248
conversion tables 376
files
associating a data file with OPEN 250
closing a PL/I file 250
creating 250
record 252
remote file access 232
representations, portability 7
structures 323
testing 211
transmission 233
types
 equivalent Java and PL/I 431
equivalent SQL and PL/I 164
data set
VSAM
file attribute 291
workstation VSAM
dummy data set 291
data sets
access methods 231
accessing
elements of REGIONAL(1) 272
record I/O 271
associating a PL/I file with a data set how PL/I finds data sets 250
using environment variables 248
using the TITLE option of the OPEN statement 249
using unassociated files 250
associating several data sets with one file 250
associating with more than one file 250
characteristics 229
combinations of I/O statements, attributes, and options 251
DD:ddname environment variable 239, 248
default identification 248
defining and using 288
disassociating 250
DISPLAY statement input and output 253
establishing a path 250
establishing characteristics
data set organizations 233
DD:ddname environment variable 239
PL/I ENVIRONMENT
attribute 233
record formats 233
records 232
extending on output 240
keyed access 231
maximum number of regions 244
native, fixed-length 230
number of regions 244
opening a PL/I file 250
organization
DDM and VSAM 239
default 233
options 233
regional 238
PL/I standard files (SYSPRINT and SYSIN) 254
record I/O access 230
recreating output 240
redirecting standard input output, and error devices 254
regional 231
REGIONAL(1) 231
sequential access 230
specifying characteristics 222
stream files 256
types
conventional text files and devices 230
fixed-length data sets 230
native data sets 230
regional data sets 231
data sets (continued)
VSAM 231
workstation VSAM
defining files 288
direct data sets 305
keyed data sets 298
organization 287
sequential data sets 295
data-directed I/O
coding for performance 361
DBCS constants 236
specifying GRAPHIC option 236
DATE attribute
definition and syntax 481
when ignored 487
DAYS built-in function 484
DAYSTODATE built-in function 485
DBCS (double-byte character set)
and GRAPHIC option 236
table names 323
DBCS compiler option 43
DBCS restrictions 6
DCLGEN 323
DD information
record format 233
TITLE statement 249
DD:ddname environment variables 239
equivalent name 249
AMTHD 239
APPEND 240
ASA 240
AUTOPAGE 241
DELAY 242
DELIMIT 242
LRECL 242
LRMSKIP 243
PROMPT 243
PUTPAGE 243
RECCOUNT 244
RECSIZE 244
RETRY 244
SAMELINE 244
SHARE 245
SKIP0 245
specifying characteristics 239
table names 323
TERMLBUF 246
RETY 246
RETURN 251
SHORT 51
debugging programs (continued)
condition handling 219
dumps 215
FLAG option 213
genereal debugging tips 212
GONUMBER option 213
NOLAXDCL option 213
NOLAXIF option 213
PREFIX option 213
RULES option 213
SNAP option 214
using compile-time options 213
using footprints for debugging
DISPLAY 215
PUT DATA 215
PUT LIST 214
PUT SKIP LIST 215
XREF option 214
DECLARE
STATEMENT definition 167
TABLE statement 167
declaring
host variables, SQL preprocessor 161
DEF files
creating 379
DEF option for ILIB 391
DEFAULT compile-time option
suboptions
DUMMY 48
LOWERINC or UPPERINC 49
DEFAULT compiler option 44
suboptions
ALIGNED 46
ANS or IBM 46
ASCII or EBCDIC 46
ASSIGNABLE or NONASSIGNABLE 46
BIN1ARG or NOBIN1ARG 46
BYADDR or BYVALUE 47
CONNECTED or NONCONNECTED 47
DESCLIST or DESCLOCATOR 47
DESCRIPTOR 47
E 48
EVENDEC or ODDENDEC 48
IEEE or HEXADEC 49
INITFILL or NONINITFILL 49
INLINE or NONINLINE 49
LINKAGE 49
NATIVE or NONNATIVE 49
NATIVEADDR or NONNATIVEADDR 50
NULLSTRADDR or NONNULLSTRADDR 50
NULLSTRPTR 50
NULLSYS or NULL370 50
ORDER or ORDER 51
ORDINAL 51
OVERLAP or NOOVERLAP 51
PSEUDODUMMY 51
RECURSIVE or NONRECURSIVE 51
RETCODE 51
RETURNS 51
SHORT 51
using default suboptions 357
Index 517
DEFAULT statement restrictions  5
DEFINED
versus UNION  364
defining files
for data sets  292
for REGIONAL(1) data sets  279
definition file
creating  379
DELAY option
description and syntax  242
DELETE statement  252
DELIMIT option
description and syntax  242
DEPRECATES compiler option  53
DEPRECATENEXT compiler option  55
DESCLIST compile-time suboption  47
DESCLOCATOR compile-time suboption  47
descriptor area, SQL  159
description file
creating  379
DELAY option
description and syntax  242
DELETE statement  252
DELIMIT option
description and syntax  242
DEPRECATES compiler option  53
DEPRECATENEXT compiler option  55
DESCLIST compile-time suboption  47
DESCLOCATOR compile-time suboption  47
descriptor area, SQL  159
deck checking  211
device
character  229
con  253
standard  229
std  253
direct access  280
DIRECT file
using to access a workstation VSAM
direct data set  309
using to access a workstation VSAM
keyed data set  302
directing I/O  253
DISPLAY  215
Distributed Data Management  229
DLLINIT compiler option  56
DLLs  379
DPATH run-time environment
variable  207
DRIVER sample program
compile, link, and run  382
example of FETCHing a DLL at run time  381
DRIVER1.DEF file
sample program to build a DLL  381
DRIVER1.PLI file
sample program that uses a DLL  381
DSNTIAR.PLI sample program  169
dummy
cards  277
DUMMY compile-time suboption  48
dummy records
workstation VSAM  291
dumps
condition handling  219
default options  216
error handling  219
formatted PL/I
dumps—PLIDUMP  216, 217
options string  215
SNAP dumps for trace
information  219
title string  216
dynamic descendancy  219
dynamic link libraries
building  379
compiling, linking, running  381
creating DLL source files  379
using FETCH and RELEASE in your
main program  381
ERROR
ON-units  12
error and condition handling
conditions used for testing and
debugging  222
dynamic descendancy  219
general concepts  219
interrupts and PL/I conditions  219
normal return  220
ON-units for conditions  222
standard system action  220
static descendancy  219
terminology  219
errors
calling uninitialized entry
variables  223
compiler or library  225
differences in issuing from PL/I for
z/OS  12
invalid use of PL/I  222
logical errors in source program  222
loops  223
poor performance  226
run-time messages  220
system failure  225
unexpected
input/output data  224
program results  225
program termination  224
unforeseen errors  223
EVERDEC compile-time suboption  48
EXEC SQL statements  150
EXIT compiler option  57
exporting data from a DLL  382
EXTDICTIONARY option for ILIB  393
EXTRACT object for ILIB  389
EXTRN compiler option  58
FETCH statement
using in your main program  381
files
adapting existing programs for
workstation VSAM
using CONSECUTIVE files  293
using INDEXED files  293
using REGIONAL(1) files  294
using VSAM files  294
closing  250
declarations for REGIONAL(1) data
sets  277
defining
record I/O  270
for workstation VSAM data
sets  292
stream I/O  257
opening  250
PL/I
definition  229
standard  254
printer-destined  256
STREAM attribute  256
SYSSIN  265
SYSPRINT  265
filespec  239
FILLERS  264
filtering messages  370
FIXED
BINARY, mapping and portability 11
TYPE option 247
fixed-length record format 233
FLAG compile-time option
using when debugging 213
FLAG compiler option 59
floating-point data 9
FLOATINMATH compiler option 60
footprints for debugging 214
formatted PL/I dumps 215
FREEFORMAT option for ILIB 392
GENDEF (/gd) option for ILIB 392
geneneral purpose register implications
example
passing conforming parameters to
a routine 398
passing floating point parameters
to a routine 400
parameters 397
generating declare statements 323
GENIMPLIB (/gi) option for ILIB 392
GENKEY option 234
GET statement 251
GET statements
controlling input from the
terminal 265
GRAPHIC option 236
global control blocks
data entry fields 372
writing the initialization
procedure 373
writing the message filtering
procedure 373
writing the termination
procedure 375
GONUMBER compile-time option
using when debugging 213
GONUMBER compiler option 61, 356
GRAPHIC
ENVIRONMENT option 236, 257
GRAPHIC compiler option 62
graphic data and stream I/O 256
H
handling conditions
built-in for condition handling 221
PL/I run-time error message
formats 220
SNAP messages 221
system messages 221
sources of conditions 220
HELLO program 17
HELP option for ILIB 392
HEXADECIMAL
compile-time suboption 49
portability considerations 9
host
structures 165
variables, using in SQL
statements 161
I
I/O
access methods
BTRIEVE 231, 232
ISAM 231
REMOTE 232
SFS 232
attributes table 251
DDM 231
options table 251
redirection 254
statements table 251
unexpected 224
using the sort program 441
IBM compile-time suboption 46
IBMUEXIT compiler exit 370
IEEE
compile-time suboption 49
portability considerations 9
IGNORE compiler option 63
ILIB
input 386
introduction 383
invoking 383
objects 390
output 386
specifying parameters 383
using a response file 385
ILINK environment variable 193
link syntax 185
IMPRECISE compiler option 64
improving performance 355
improving application performance 355
INCAFTER compiler option 65
INCLUDE
environment variable 24
processing 19
statement, using DCLGEN 327
INCLUDE compiler option 66
include files
with ODBC 319
include preprocessor
environment variables 148
syntax 147
INDEXAREA ENVIRONMENT option 4
indexed data sets 231
INDEXED files, adapting programs 293
indicator variables, SQL 166
INITAUTO compiler option 67
INITBASED compiler option 68
INITCTL compiler option 69
INITFILL compile-time suboption 49
INITIAL attribute 11
initialization procedure of compiler user
exit 373
INITSTATIC compiler option 70
INLINEx compile-time suboption 49
input
controlling from terminal 265
defining data sets for stream
files 256
example of interactive program 268
SEQUENTIAL 269
to the terminal
example of an interactive
program 268
input (continued)
to the terminal (continued)
format of PRINT files 268
stream and record files 268
input and output with workstation
VSAM data sets
description 287
organization
accessing records in 288
creating and accessing 288
determining which type you
need 288
using keys 289
using workstation VSAM direct data
sets
loading 307
using a DIRECT file to access 309
using a SEQUENTIAL file to
access 309
using workstation VSAM keyed data
sets
loading 300
using a DIRECT file to access 302
using a SEQUENTIAL file to
access 302
using workstation VSAM sequential
data sets
defining and loading 296
updating 298
using a SEQUENTIAL file to
access 296
input- and output-handling routines, sort
program 441
INSOURCE compiler option 71
Institute (ANSI)
in printer-destined files 255
interactive program, example 268
interlanguage communication (ILC) 411
interrupts 213
invoking
compiler 24
ISAM access method 231
isUB defining restrictions 6
J
Java 420, 421, 422, 423, 424, 425, 426,
427, 429, 430, 431
JAVA 419
Java code, compiling 421, 424, 429
Java code, writing 420, 423, 427
jni
JNI sample program 420, 423, 427
K
key
accessing a sequential data set 290
generic 234
relative record number
padding 290
truncation 290
starting position 236
KEY
keys for workstation VSAM keyed
data sets 289
Index 519
KEY (continued)
relative record numbers 290
sequential record values 290
key length, checking 236
keyboard
screen operations 253
keyed data sets
statements and options for 298
types and advantages 289
KEYFROM 289
KEYFROM, relative record numbers 290
KEYLENGTH option 236
KEYLOC option 236
keys
using for workstation VSAM keyed
data sets 289
using relative record numbers 290
using sequential record values 290
KEYTO
keys for workstation VSAM keyed
data sets 289
relative record numbers 290
sequential file to access a workstation
VSAM sequential data set 296
sequential record values 290
L
LANGLVL compiler option 72
large object (LOB) support, SQL
preprocessor 167
LEAVE ENVIRONMENT option 4
length of record
maximum 237
specifying 244
library manager 383
library, compiler subroutine failure 225
LIBS compiler option 73
LIMITS compiler option 74
line continuation 20
line feed (LF)
definition 247
delimiting logical records 230
LF files 232
LINE option
in controlling output to the
terminal 268
of PUT statement 255
using with PRINT files 261
when using PRINT files 261
LINECOUNT compiler option 75
LINEDIR compiler option 76
LINESIZE option
accessing a data set with stream
I/O 257
creating a data set with stream
I/O 257
definition 257
OPEN statement 257
tab set table field 264
linkage
OPTLINK
example 397
features 396
tips for using 397
SYSTEM
description 403
linkage (continued)
SYSTEM (continued)
example 404
LINKAGE compile-time suboption
calling conventions 395
effect on performance 359
syntax 49
linking your program
creating files
dynamic link library 190
executable files 190
map 191
input and output 187
return codes 191
search rules 188
specifying directories 188
starting the linker 185
static linking 185
using a make file 186
using response files 189
using the command line 185
lowercase inc element
of PUT statement 255
mapping file (MAP) 260
macros
description 260
specifying 260
macro facility
environment variables 150
IBM.PPMACRO 22
macro definition 148
portability 7
macro preprocessor
macro definition 148
mainframe applications
running on the workstation 7
make file utility (NMAKE) 333
managing libraries (LIB files) 383
MARGINI compiler option 81
margins 20
MARGINS compiler option 82
MAXGEN compiler option 83
MAXMSG compiler option 84
MAXNEST compiler option 85
MAXSTMT compiler option 86
MAXVAR compiler option 87
MDECK compiler 
messages
filter function 373
modifying in compiler user exit 371
migration
compatibility with PL/I for z/OS 3
migration (continued)
PL/I for z/OS files for the
workstation
CONSECUTIVE file 293
EXCLUSIVE file 293
INDEXED file 293
ISAM record handling 293
REGIONAL(1) file 294
VSAM file 294
Millennium Language Extensions
using in PL/I applications 481
mixed-language applications 411
module testing 212
MSG compiler option 89
MSGSUMMARY compiler option 90
N
named constants
defining 365
versus static variables 365
NAMES compiler option 91
national characters 3
national language support 208
NATIVE compile-time suboption
description 49
effect on performance 360
portability considerations 8
native data sets
accessing 230
character devices 230
c conventional text files 230
DDM data sets 231
fixed-length data sets 230
regional data sets 231
types 230
NATIVEADDR compile-time
suboption 50
NATLANG
run-time option 208
NATLANG compiler option 92
NCP ENVIRONMENT option 4
NEST compiler option 93
NMAKE utility
characters that modify
commands 352
descriptions 337
directives 347
inference rules 345
inline files 350
introduction 333
make file utility (NMAKE) 333
make file utility (NMAKE) 333
make file utility (NMAKE) 333
special macros 342
syntax 333
TOOLS.INI file 354
using command files 334
using macros 340
using the command line 333
NOBACKUP option for ILIB 391
NOBIN1ARG compiler option 46
NODESCRIPTOR compile-time
suboption 47
NOEVENDEC compile-time
suboption 48
NOEXTDICTIONARY option for
ILIB 393
RESPECT compiler option 117, 482
RETCODE compile-time suboption 51
RETRY option 244
return codes, linker 191
RETURNS compile-time suboption 51, 359
REWITE statement 252
REWITE statement, attributes and options 253
routines, library, conversions 367
RULES compile-time option using when debugging 213
RULES compiler option 118, 482
effect on performance 356
RULES output options 7
SNAP 221
SYSTEM 221
options, specifying 207
shipping DLLs 209
NATLANG 208
specifying multiple run-time options or suboptions 208
where to specify run-time options 207
running your program
setting run-time environment variables 207
specifying run-time options 207
runtime behavior differences
ERROR message issuing 12
INITIAL attribute for AREAs is ignored 11
language elements 11
using variables declared as FIXED BIN 11
differences between platforms 7
SAMELINE option 244
sample program, running 423, 427, 431
sample programs 18
SCALARVARYING option 238
screen and keyboard operations 253
search rules
linker 188
SEMICANTic compiler option 126
sequential
access 280
data sets
statements and options 295
workstation VSAM 289
record value
in workstation VSAM sequential data set 290
using KEYTO to find 296
SEQUENTIAL OUTPUT 270
SEQUENTIAL file
using to access a workstation VSAM direct data set 309
using to access a workstation VSAM keyed data set 302
statement numbers, determining from offset 221
statements
DELETE 252
GET 251
LOCATE 251
READ 252
REWITE 252
WRITE 251
STATIC compiler option 130
static descendancy 219
static linking 185
STMT compiler option 131
storage
report in listing 132
STORAGE compiler option 132
stream and record files 268
STREAM attribute
data sets 256
discussion 256
stream I/O
accessing data sets 259
creating a data set 257
essential information 257
element 258
STREAM I/O restrictions 4
stream-oriented data transmission
accessing a data set with stream I/O essential information 260
example 260
creating a data set with stream I/O 257
defining files using stream I/O 256
ENVIRONMENT options for stream-oriented data transmission 257
using PRINT files 261
using SYSIN and SYSPRINT files 265
structure expression restrictions 5
structure of global control blocks
writing the initialization procedure 373
writing the message filtering procedure 373
writing the termination procedure 375
subtracting dates 486
SYNTAX compiler option 133
SYSIN files attributes 265
redirecting standard input 254
SYSPARM compiler option 134
SYSPRINT files attributes 265
redirecting standard output 254
system
error-handling facilities 220
failure 225
messages 221
standard action for conditions 220
SYSTEM linkage, calling conventions 403
message 221
SYSTEM compiler option 135
tab control table 264
terminal
   conversational I/O 266
example of an interactive program 268
input 265
output 268
TERMINAL compiler option 136
termination procedure
   compiler user exit 375
example of procedure-specific control block 375
syntax
   global 372
   specific 375
TERMLBUF option 246
TEST compiler option 137
testing programs
   code inspection 211
data testing 211
   path testing 212
text file
   conventional 229
   LF 229
TITLE option
   description 249
   opening and closing a file 250
   specifying the length of records 232
   using
   RECSIZE option 257
   SYSPRINT and SYSIN files 254
   using files not associated with data sets 250
TMP environment variable 24
TOTAL ENVIRONMENT option 4
TP ENVIRONMENT option 4
trace information 215
TRKOFI ENVIRONMENT option 4
TYPE option 246
   specifying record formats 233
U
   U-format record 233
   undefined-length record format 233
UNDEFINEDFILE condition
   raising when opening a file 248
   using files not associated with data sets 248
unexpected
   input/output data 224
   program end 224, 225
   uninitialized entry variables 223
UNLOCK statement 4
UPPERINC compile-time suboption 49
USAGE compiler option 138
user defined functions, SQL
preprocessor 168
user exit
   CICS run-time 375
   compiler 369
   customizing
   modifying IBMUEXIT.INF 371
   structure of global control blocks 372
user exit (continued)
   customizing (continued)
   writing your own compiler exit 372
   functions 370
   using
   arrays as host variable, SQL
   preprocessor 161
   using host variables, SQL
   preprocessor 161
Using Millennium Language Extensions
   date patterns 483
   language features 481
   using the sort program 433
V
   variables
   environment variables for compile time 21
   varying-length records
   format 233
   sorting 441
   VSAM files, adapting programs 294
   option 239
W
   WARN option for ILIB 393
   WIDECHAR compiler option 139
   WINDOW compile-time option 140
   WINDOW compiler option 140, 482
workstation
   native data sets 229
   record format 233
   text file 230
workstation VSAM (virtual storage access method)
   data sets
   dummy data set 291
   file attribute 291
workstation VSAM data sets
   accessing records 288
   adapting programs
   using CONSECUTIVE files 293
   using INDEXED files 293
   using REGIONAL(1) files 294
   using VSAM files 294
   choosing a type 290
   defining files
   adapting existing programs 293
   specifying options of the PL/I ENVIRONMENT attribute 292
   direct 288
   file declaration 292
   keyed
   base file 288
   prime index file 288
   sequential 288
types and advantages 288
workstation VSAM direct data sets
   loading 307
   using a DIRECT file to access 302
workstation VSAM keyed data sets
   loading 300
   using a DIRECT file to access 302
   using a SEQUENTIAL file to access 302
workstation VSAM sequential data sets
   accessing from a SEQUENTIAL file 296
   defining and loading 296
   updating 298
   using a SEQUENTIAL file to access 296
WRITE statement, attributes and options 251
X
   XINFO compiler option 141
   XML compiler option 144
   XREF compile-time option
   output in listing 181
   using when debugging 214
Readers’ Comments — We’d Like to Hear from You

Rational Developer for System z, Version 8.5
PL/I for Windows
Programming Guide
Version 8.5

Publication No. SC18-9977-05

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