IBM PL/I for AIX

Programming Guide

Version 2.0.0

SC18-9328-00

IBM PL/I for AIX

Programming Guide

Version 2.0.0

SC18-9328-00

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 309.

Second Edition (June 2004)

This edition applies to IBM PL/I for AIX 2.0.0, 5724-H45, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

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

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

IBM Corporation, Department HHX/H1
555 Bailey Ave
San Jose, CA, 95141-1099
United States of America

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

Contents

Figures vii

Part 1. Introducing PL/I on your workstation 1

Chapter 1. About this book 3

Chapter 2. How to read the syntax diagrams 5

Chapter 3. Porting applications between platforms 9

Getting mainframe applications to compile on the workstation. 9
- Choosing the right compile-time options . . . 10
- Language restricted. 10
- Using the macro facility to help port programs 13

Getting mainframe applications to run on the workstation 14
- Data representations causing run-time differences 14
- Environment differences affecting portability . . 16
- Language elements causing run-time differences 17

Part 2. Compiling and linking your program 19

Chapter 4. Introducing PL/I for AIX. . . 21

Introducing PL/I for AIX. 21
- Features available with PL/I for AIX 21
- A short practice exercise 22
- Becoming familiar with AIX 22
- Creating PL/I source programs. 24

Chapter 5. Compiling, linking, and running your program 29

Compiling, linking, and running your program . . 29
- Invoking the PL/I for AIX compiler 29
- Specifying compile-time options 35
- Invoking the linkage editor 38
- Running a program. 38

Chapter 6. Compile-time option descriptions 41

Compile-time option descriptions 41
Rules for using compile-time options 43
- AGGREGATE 44
- ATTRIBUTES. 44
- BIFPREC 45
- BLANK. 46
- CHECK 46
- CMPAT. 46
- CODEPAGE 47
- COMPILE 47
- COPYRIGHT 48
- CURRENCY 48
- DEFAULT 48
- EXIT. 54
- EXTRN 54
- FLAG 55
- FLOATINMATH. 55
- GONUMBER 56
- GRAPHIC 56
- IMPRECISE 56
- INCAFTER 57
- INITAUTO 57
- INITBASED 57
- INITCTL 58
- INITSTATIC 58
- INCDIR 58
- INSOURCE 58
- LANGLVL. 59
- LIMITS 60
- LINECOUNT. 60
- LIST. 61
- MACRO 61
- MARGINI 61
- MARGINS. 62
- MAXMEM. 62
- MAXMSG 63
- MAXSTMT 64
- MAXTEMP 64
- MDECK 64
- MSG. 64
- NAMES 65
- NATLANG 65
- NEST 65
- NOT. 66
- NUMBER 66
- OPTIMIZE. 67
- OPTIONS 67
- OR 67
- PP 68
- PPTRACE 69
- PRECTYPE 69
- PREFIX. 69
- PROCEED. 70
- REDUCE 71
- RESEXP 71
- RESPECT 72
- RULES 72
- SEMANTIC 75
- SOURCE 76
- STATIC. 76
- SPILL 76
- STMT 77
- STORAGE. 77
- SYNTAX 77
- SYSPARM 78

SYSTEM 78
TERMINAL 78
TEST 79
USAGE 79
WIDECHAR 79
WINDOW 80
XINFO 80
XREF 81

Chapter 7. PL/I preprocessors 83
Include preprocessor 84
Examples: 84
Include preprocessor options in the configuration file 84
Macro preprocessor 85
Macro preprocessor options 85
Macro facility options in the configuration file . . 86
SQL support 87
Programming and compilation considerations . . 87
SQL preprocessor options 87
Abbreviations: 89
SQL preprocessor options in the configuration file 92
Coding SQL statements in PL/I applications . . 93
Large Object (LOB) support 98
User defined functions sample programs . . . 100
CICS support 107
Programming and compilation considerations 107
CICS preprocessor options 108
Abbreviations: 109
Coding CICS statements in PL/I applications 109
Writing CICS transactions in PL/I 110

Chapter 8. Compilation output 111
Compilation output 111
Using the compiler listing 111
Compiler output files 121

Part 3. Running and debugging your program 123

Chapter 9. Testing and debugging your programs 125
Testing your programs 125
General debugging tips 126
PL/I debugging techniques 127
Using the Distributed Debugger tool 127
Using compile-time options for debugging . . 127
Using footprints for debugging 128
Using dumps for debugging 129
Using error and condition handling for debugging 133
Error handling concepts 134
Common programming errors 136
Logical errors in your source programs 136
Invalid use of PL/I 137
Calling uninitialized entry variables 137
Loops and other unforeseen errors 137
Unexpected input/output data 138
Unexpected program termination 138
Other unexpected program results 139
Compiler or library subroutine failure 139
System failure 140
Poor performance 140

Part 4. Input and output 141

Chapter 10. Using data sets and files 143
Types of data sets 143
Native data sets 144
Additional data sets 145
Establishing data set characteristics 146
Records 146
Record formats 146
Data set organizations 146
Specifying characteristics using the PL/I ENVIRONMENT attribute 147
Specifying characteristics using DD_ddname environment variables 152
Associating a PL/I file with a data set 161
Using environment variables 162
Using the TITLE option of the OPEN statement 162
Attempting to use files not associated with data sets 163
How PL/I finds data sets 163
Opening and closing PL/I files 163
Opening a file 163
Closing a file 163
Associating several data sets with one file 164
Combinations of I/O statements, attributes, and options 164
DISPLAY statement input and output 166
PL/I standard files (SYSPRINT and SYSIN) . . . 167
Redirecting standard input, output, and error devices 167

Chapter 11. Defining and using consecutive data sets 169
Printer-destined files 169
Using stream-oriented data transmission 170
Defining files using stream I/O 171
ENVIRONMENT options for stream-oriented data transmission 171
Creating a data set with stream I/O 171
Accessing a data set with stream I/O 173
Using PRINT files 175
Using SYSIN and SYSPRINT files 179
Controlling input from the console 180
Using files conversationally 181
Format of data 181
Stream and record files 181
Capital and lowercase letters 182
End of file 182
Controlling output to the console 182
Format of PRINT files 182
Stream and record files 182
Example of an interactive program 182
Using record-oriented I/O 183
Defining files using record I/O 184

ENVIRONMENT options for record-oriented data transmission 185
Creating a data set with record I/O 185
Accessing and updating a data set with record I/O. 185

Chapter 12. Defining and using regional data sets 191
Defining files for a regional data set. 193
Specifying ENVIRONMENT options 193
Essential information for creating and accessing regional data sets 194
Using keys with regional data sets 194
Using REGIONAL(1) data sets 194
Dummy records 194
Creating a REGIONAL(1) data set 195
Example 195
Accessing and updating a REGIONAL(1) data set 197
Sequential access 197
Direct access. 198
Example 198

Chapter 13. Defining and using workstation VSAM data sets 201
Remote file access 201
Workstation VSAM organization 202
Creating and accessing workstation VSAM data sets. 202
Determining which type of workstation VSAM data set you need 202
Accessing records in workstation VSAM data sets. 202
Using keys for workstation VSAM data sets . . 203
Choosing a data set type 204
Defining files for workstation VSAM data sets . . 204
Specifying options of the PL/I ENVIRONMENT attribute 205
Adapting existing programs for workstation VSAM. 205
Using workstation VSAM sequential data sets . . 208
Using a sequential file to access a workstation VSAM sequential data set 209
Defining and loading a workstation VSAM sequential data set. 209
Workstation VSAM keyed data sets 211
Loading a workstation VSAM keyed data set 213
Using a SEQUENTIAL file to access a workstation VSAM keyed data set 215
Using a DIRECT file to access a workstation VSAM keyed data set 215
Workstation VSAM direct data sets 218
Loading a workstation VSAM direct data set 220
Using a SEQUENTIAL file to access a workstation VSAM direct data set 222
Using a DIRECT file to access a workstation VSAM direct data set. 223

Part 5. Advanced topics. 227

Chapter 14. Using user exits 229
Using the compiler user exit 229
Procedures performed by the compiler user exit 229
Activating the compiler user exit 230
The IBM-supplied compiler exit, IBMUEXIT . . 230
Customizing the compiler user exit 231
Using data conversion tables 234

Chapter 15. Improving performance 237
Selecting compile-time options for optimal performance. 237
OPTIMIZE 237
IMPRECISE 238
GONUMBER 238
RULES 238
PREFIX 239
DEFAULT 239
Summary of compile-time options that improve performance. 242
Coding for better performance 242
DATA-directed input and output 243
Input-only parameters 243
String assignments 243
Loop control variables 244
PACKAGEs versus nested PROCEDUREs . . . 244
REDUCIBLE functions 245
DEFINED versus UNION 246
Named constants versus static variables . . . 246
Avoiding calls to library routines. 247

Chapter 16. Using PL/I in mixed-language applications. 251
Matching data and linkages 251
What data is passed 252
How data is passed 253
Where data is passed. 255
Maintaining your environment 255
Invoking non-PL/I routines from a PL/I MAIN 255
Invoking PL/I routines from a non-PL/I main . . 256
Using ON ANYCONDITION 256

Chapter 17. Using sort routines . . . 259
Comparing S/390 and workstation sort programs 259
Preparing to use sort 260
Choosing the type of sort 261
Specifying the sorting field 263
Specifying the records to be sorted 264
Calling the sort program 264
PLISRT examples 264
Determining whether the sort was successful 265
Sort data input and output 266
Sort data handling routines. 266
E15 — input-handling routine (sort exit E15) 266
E35 — output-handling routine (sort exit E35) 269
Calling PLISRTA 271
Calling PLISRTB 272
Calling PLISRTC 274
Calling PLISRTD, example 1 275
Calling PLISRTD, example 2 276

Chapter 18. Using the SAX parser . . 277
Overview. 277
The PLISAXA built-in subroutine. 278
The PLISAXB built-in subroutine. 278
The SAX event structure. 278
start_of_document. 279
version_information 279
encoding_declaration. 279
standalone_declaration 279
document_type_declaration. 279
end_of_document 279
start_of_element 280
attribute_name 280
attribute_characters 280
attribute_predefined_reference. 280
attribute_character_reference 280
end_of_element. 280
start_of_CDATA_section. 280
end_of_CDATA_section 281
content_characters. 281
content_predefined_reference 281
content_character_reference. 281
processing_instruction 281
comment. 281
unknown_attribute_reference 282
unknown_content_reference 282
start_of_prefix_mapping. 282
end_of_prefix_mapping 282
exception. 282
Parameters to the event functions 282
Coded character sets for XML documents 283
Supported EBCDIC code pages 283
Supported ASCII code pages 284
Specifying the code page 284
Exceptions 285
Example 286
Continuable exception codes 297
Terminating exception codes 301

Part 6. Appendixes 305

Product specifications and service 307
Product specifications and service 307
Specified operating environment 307
Installing PL/I for AIX 307
Shipping shared runtime libraries 307

Notices 309
Programming interface information 310
Macros for customer use 310
Trademarks 311

Bibliography. 313
Enterprise PL/I publications 313
DB2 UDB for OS/390 and z/OS 313
CICS Transaction Server. 313

Glossary 315

Index 329

Figures

1. Sample configuration file 33
2. The PL/I declaration of SQLCA. 93
3. The PL/I declaration of an SQL descriptor area 94
4. TOWERS program compiler listing 111
5. PL/I code that produces a formatted dump 131
6. Example of PLIDUMP output 132
7. Static and dynamic descendant procedures 134
8. Creating a data set with stream-oriented data transmission 173
9. Accessing a data set with stream-oriented data transmission 175
10. Creating a print file via stream data transmission 177
11. Declaration of PLITABS 178
12. PL/I structure PLITABS for modifying the preset tab settings 179
13. A sample interactive program 183
14. Merge Sort—Creating and accessing a consecutive data set 187
15. Printing record-oriented data transmission 190
16. Creating a REGIONAL(1) data set 196
17. Updating a REGIONAL(1) data set 199
18. Creating a workstation VSAM keyed data set 207
19. Defining and loading a workstation VSAM sequential data set 210
20. Defining and loading a workstation VSAM keyed data set 214
21. Updating a workstation VSAM keyed data set 216
22. Loading a workstation VSAM direct data set 221
23. Updating a workstation VSAM direct data set by key 224
24. PL/I compiler user exit procedures 230
25. Example of an IBMUEXIT.INF file 231
26. Flow of control for the sort program 262
27. Skeletal code for an input procedure 268
28. When E15 is external to the procedure calling PLISRTx 269
29. Skeletal code for an output-handling procedure. 270
30. PLISRTA—Sorting from input data set to output data set 271
31. PLISRTB—Sorting from input-handling routine to output data set 272
32. PLISRTC—Sorting from input data set to output-handling routine 274
33. PLISRTD—Sorting input-handling routine to output-handling routine 275
34. PLISRTD—Sorting input-handling routine to output-handling routine 276
35. Sample XML document 279
36. PLISAXA coding example - type declarations 286
37. PLISAXA coding example - event structure 287
38. PLISAXA coding example - main routine 288
39. PLISAXA coding example - event routines 289
40. PLISAXA coding example - program output 297

Part 1. Introducing PL/I on your workstation

Chapter 1. About this book

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

If you have typically used mainframe PL/I and are interested in moving your programs to the AIX platform, Chapter 3, “Porting applications between platforms,” on page 9 should be particularly useful. Other information in this guide is designed to help you understand basic AIX features as well as compile, link, and run a PL/I program using the PL/I for AIX compiler.

A number of chapters on PL/I input and output are included as well as general debugging information. Since a lot of system information is available in existing documentation, it is not repeated in this guide. For the complete titles and order numbers of related publications, see the “Bibliography” on page 313.

Chapter 2. 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.

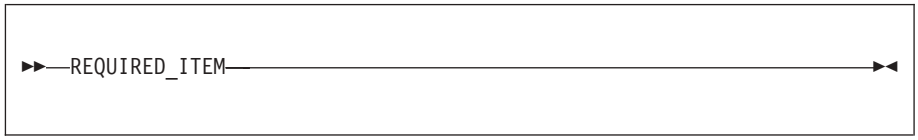
Diagrams of syntactical units other than complete statements start with the >--- symbol and end with the --->

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.

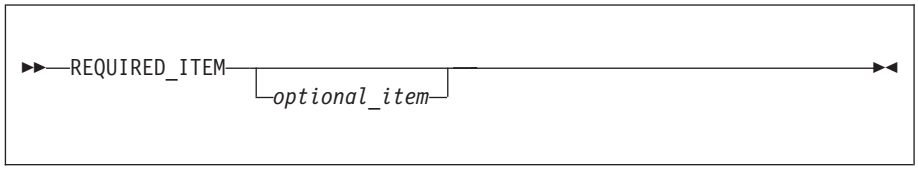
Required items

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

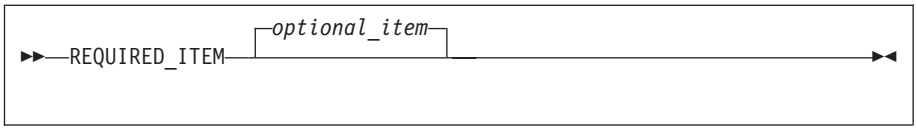


Optional Items

Optional items appear below the main path.

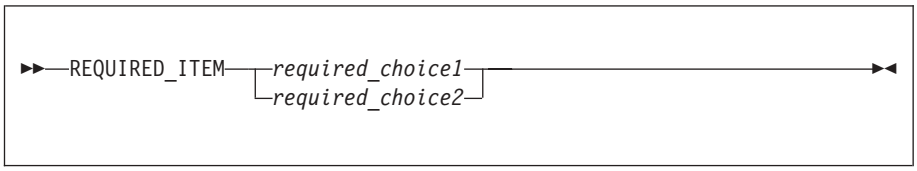


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

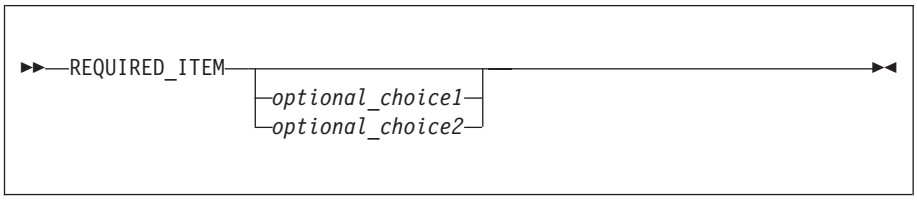


Multiple required or optional items

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

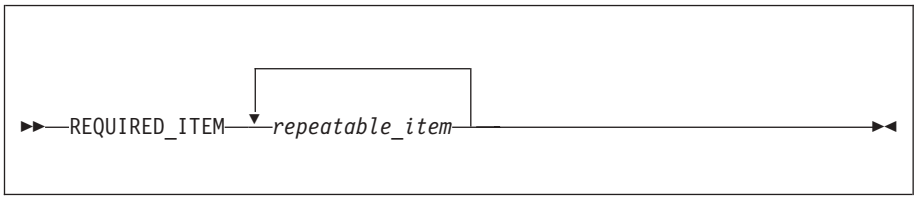


If choosing one of the items is optional, the entire stack appears below the main path.

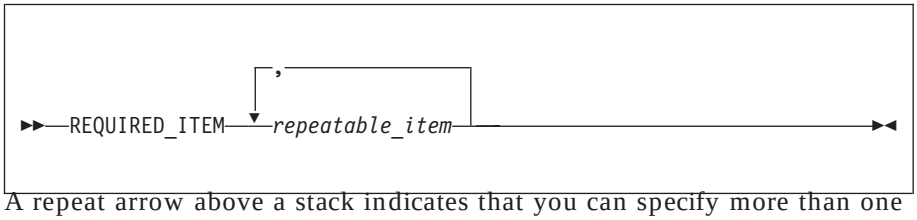


Repeatable items

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



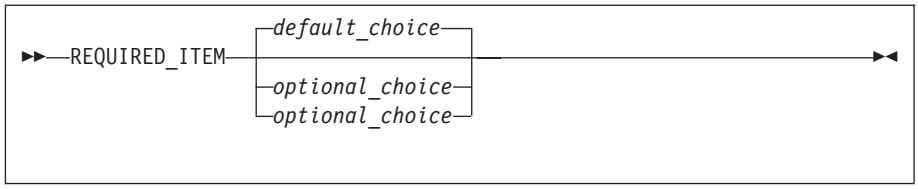
If the repeat arrow contains a comma, you must separate repeated items with a comma.



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

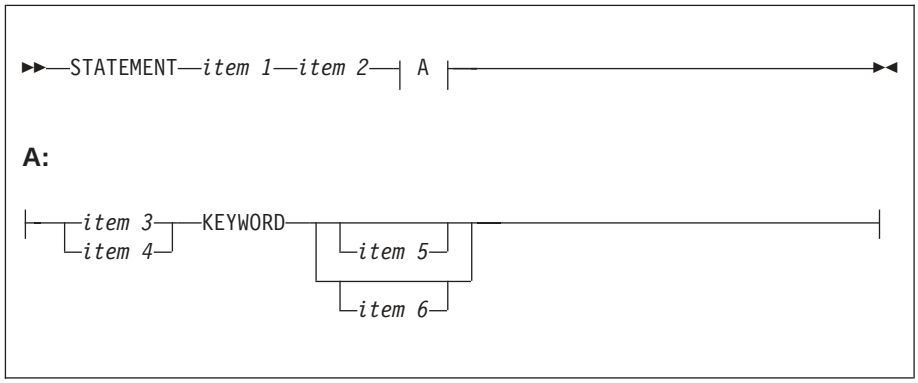
Default keywords

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.



Chapter 3. Porting applications between platforms

Getting mainframe applications to compile on the workstation 9
Choosing the right compile-time options . . . 10
Language restricted 10
RECORD I/O 10
STREAM I/O 11
Structure expressions 11
Array expressions 11
DEFAULT statement 11
Extents of automatic variables 12
Built-in functions 12
iSUB defining 12
DBCS 12
Macro preprocessor 12
Using the macro facility to help port programs 13
Getting mainframe applications to run on the workstation 14
Data representations causing run-time differences 14
Environment differences affecting portability . . 16
Language elements causing run-time differences 17

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.

Because of fundamental platform differences as well as difference the OS PL/I compiler and the PL/I for Windows compilers, some problems can arise as you move PL/I programs between the mainframe and workstation environments. 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 hex 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 hex 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 hex editor, some of those values might have a hex 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” (|) 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 66), OR (see “OR” on page 67), and CURRENCY (see “CURRENCY” on page 48) compile-time options on the *PROCESS statement. Avoid potential problems involving other characters by using the NAMES (see “NAMES” on page 65) compile-time option to define extramural characters and symbols.

File names

File names in the AIX environment are generally case sensitive. PL/I for AIX, however, accepts both uppercase and lowercase filenames for the main source file. By default, the filename for INCLUDE files must be in all lowercase, but the UPPERINC suboption of the DEFAULT compile-time option (see 52) allows you to specify INCLUDE filenames in all uppercase.

Choosing the right compile-time options

By selecting certain compile time options, you can make your source code more portable across compilers and platforms. For instance, if you select LANGLVL(SAA), the compiler flags any keywords not supported by pre-Enterprise PL/I and does not recognize any built-in functions not supported by pre-Enterprise PL/I.

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

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

Note that the option DEFAULT(RETURNS(BAYDDR)) 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 6, “Compile-time option descriptions,” on page 41 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 file attributes are not supported:
  - BACKWARDS
  - EXCLUSIVE
  - TRANSIENT
- The following options of the ENVIRONMENT attribute are not supported, but their use is flagged only under LANGLVL(NOEXT):
  - ADDBUFF
  - ASCII
  - BUFFERS
  - BUFND

- BUFNI
- 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.
- 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:

```
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:

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

call a1(b || c);
```

However, the following prototyped call would not be flagged:

```
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:

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

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

```
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:
  - 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 OS PL/I Version 2 suffixes—unless you insert a delimiter between the ending quotation mark of the string and the first letter of the suffix.

Note that the OS PL/I V2R1 compiler introduced this change, and so this is not a difference between the PL/I for MVS & VM compiler and either the PL/I for MVS & VM compiler or the OS PL/I V2Rx compilers. This restriction is consequently not flagged.

As an example, consider:

```
%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:

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

whereas, under PL/I for MVS & VM it produces:

```
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 AIX macro facility COMPILETIME built-in function returns the date using the format 'DD.MMM.YY', while the OS PL/I 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 AIX and all versions of the mainframe PL/I compiler, for example:

```
%dcl compiletime builtin;

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

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

Getting mainframe applications to run 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.

Data representations causing run-time 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
- DEFAULT(E(HEXADEC)) instead of DFT(E(IEEE))

For more information on these compile-time options, see “DEFAULT” on page 48.

ASCII vs. 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 234.

NATIVE vs. 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 two or more bytes
- OFFSET variables
- The length prefix of VARYING strings
- 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:

```
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 vs. 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 vs. normalized hexadecimal

Specification	IEEE (AIX)	IEEE (PC)	Hexadecimal

Table 1. Normalized IEEE vs. normalized hexadecimal (continued)

Approximate range of values	±10E-308 to ±10E+308	±3.30E-4932 to ±1.21E+4932	±10E-78 to ±10E+75
Maximum precision for FLOAT DECIMAL	32	18	33
Maximum precision for FLOAT BINARY	106	64	109
Maximum number of digits in FLOAT DECIMAL exponent	4	4	2
Maximum number of digits in FLOAT BINARY exponent	5	5	3

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/I should have the attributes FLOAT DEC(1), is a valid hexadecimal short float, it is not a valid IEEE short float.

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 because of the differences in how the floating-point values are represented. This is particularly true for short and extended floating-point.

EBCDIC DBCS vs. 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 for AIX are different from those on the mainframe. The following file name, for example, is valid on AIX but not on the mainframe:

```
/users/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”)

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 run-time differences

There are also some language elements that can cause your program to run differently under PL/I for Windows. than it does under OS PL/I, due to differences in the implementation of the language by the compiler. Each of the following items is described in terms of its PL/I for Windows behavior.

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

If you have any variables declared as FIXED BIN with a precision of 7 or less, they occupy one byte of storage under PL/I for Windows. instead of two as under OS PL/I. 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 product 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 desired.

```
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 as 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:

```
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 4. Introducing PL/I for AIX

Introducing PL/I for AIX

Features available with PL/I for AIX 21
A short practice exercise 22
The hello program 22
Using other sample programs 22
Becoming familiar with AIX 22
File specification. 22
Device names. 23
Setting environment variables 23
Creating PL/I source programs. 24
Program file structure 24
Program file format. 26

The following two sections summarize characteristics of the PL/I for AIX compiler and supporting run-time environment. For specific information regarding portability with pre-Enterprise PL/I see Chapter 3, “Porting applications between platforms,” on page 9.

Features available with PL/I for AIX

Strong set of language constructs

A large set of language features is supported by the compiler, many from the ANS 87 PL/I definition. These features, documented in the *PL/I Language Reference*, enhance the flexibility, power, and elegance with which you can develop programs.

Preprocessing capabilities

Throughout program development and testing, you can take advantage of the seamless integration of preprocessors. Make full use of your DB2 for AIX data by using SQL embedded in your PL/I applications. You can also use the CICS preprocessor, PL/I macro facility, or include preprocessor.

IBM Program Builder

Use this window-based tool for compiling source files, correcting source files, and linking the resulting object files into a program or library. The Program Builder supports the C, C++, FORTRAN, COBOL, and PL/I languages and offers the following features:

- **Easy-to-use interface**--The Motif-based interface provides menus and windows that simplify setting and saving compile options, linking libraries, and invoking compilers.
- **Saving the current configuration**--You can save your build and makefile options, so when you return to the same directory, your saved configuration is restored.
- **Error browsing**--Select a compilation error from the output list and automatically go to the line in the source code where the error occurred. You can also navigate through the list of errors.
- **Automatic makefile generation**-- Create makefiles for your programs or libraries and keep track of file dependency information.

Exception handling

Produce mission-critical applications that provide nonstop operation for your end users with exception handling enhancements such as ANYCONDITION, STORAGE, INVALIDOP, and RESIGNAL.

A short practice exercise

Try compiling, linking, and running a simple program to get an idea of how PL/I functions in the AIX 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, `hello1.pli`, with the following PL/I statements.

```
hello1: proc options(main);
       display('Hello!');
end hello1;
```

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 62.)

Save the file.

2. Compile and link the program

Make sure you are in the directory that contains the `hello1.pli` file and enter the following command:

```
pli hello1
```

The **pli** command compiles and links the program in one step, using default options. For details on using compile-time options and linking options, see "Compiling, linking, and running your program" on page 29.

The compiling step displays information about the compilation on your screen, and creates the object file (`hello1.o`) in the current directory. The linking step combines `hello1.o` with needed library files producing the executable file (`hello1`) in the same directory.

4. Run the program

Without changing directories, enter the following command:

```
hello1
```

This invokes the executable program and displays `Hello!` on your monitor.

Using other sample programs

Several sample programs have been included with PL/I for AIX, some of which appear in different parts of this book. The sample programs are installed in `/usr/lpp/pli/samples`.

Becoming familiar with AIX

Programming with this release of PL/I involves using some basic AIX features. These are summarized in the following sections. More details about the AIX environment are described in the AIX documentation listed in "Bibliography" on page 313.

File specification

Here is an example of how to specify a file in AIX:

```
/usr/plidev/instock.dat
```

The maximum length of a file name (the full path name) is 1023 chars. The maximum length of a file with no path is 255 characters.

Device names

The keyboard, screen, and printer (character devices) are known by the following names:

/dev/null/
: 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)

stdin:, stdout:, and stderr: can be redirected, whereas `/dev/null/` cannot.

Setting environment variables

There are a number of environment variables that can be set and exported for use with the AIX operating system.

You can set the environment variables from the Bourne shell, Korn shell, or C shell. To determine which shell is in use, issue the AIX **echo** command:

```
echo $SHELL
```

The Bourne shell path is `/bin/sh`, Korn shell is `/bin/ksh`, and C shell is `/bin/csh`.

To set the environment variables system wide in either the Bourne shell or Korn shell, so all users who use Bourne or Korn shell have access to them, add the lines suggested in the subsections to the file `/etc/profile`. To set them for a specific user only, add them to the file `.profile` in the user's home directory. The variables are set the next time the user logs on.

In the C shell, you cannot set the environment variables system wide. To set them for a specific user, add the suggested lines to the file `.cshrc` in the user's home directory. The variables are set the next time the user logs in. For more information about setting environment variables, see *AIX Commands Reference*.

The following examples illustrate how to set environment variables from various shells:

- From the Bourne or Korn shell

  ```
  LANG=ja_JP
  NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
  LIBPATH=/home/joe/usr/lib:/home/joe/mylib:/usr/lib
  export LANG NLSPATH LIBPATH
  ```

  Rather than using the last statement in the previous example, you could have added `export` to each of the preceding lines (`export LANG=ja_JP...`).

- From the C shell

  ```
  setenv LANG ja_JP
  setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
  setenv LIBPATH /home/joe/usr/lib:/home/joe/mylib:/usr/lib
  ```

Creating PL/I source programs

To create PL/I source programs, you can use a variety of text editors. If you do not specify an extension on your source file, the compiler assumes `.pli` is the extension.

For the PL/I source program to be a valid program, it must conform to the language definition specified in the *PL/I Language Reference*. You should also 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. The filename of any include file must be in lowercase. You do not compile the include files separately because they actually become part of the main program during compilation.

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::

```
%PROCESS   ;
%PROCESS   ;
%PROCESS   ;

 /*  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 statement syntax, see the *PL/I Language Reference*. The extension of all include files must be `inc`..

The name of the actual include file residing on AIX must be in lowercase. For example, if you used the include statement `%include sample`, the compiler would find the file `sample.inc`, but would not find the file `SAMPLE.inc`. Even if you used the include statement `%include SAMPLE`, the compiler would still look for `sample.inc`.

The compiler searches for include files in the following order:
1. Directories specified with the `-I` flag or `INCDIR` compile-time option
2. The /usr/include/ directory
3. Current directory

The first file found by the compiler is included into your source.

For compatibility with the OS PL/I compiler, you can specify a data set before the file name in an %INCLUDE statement, but the compiler ignores the data set specification. For example, the compiler treats the following two %INCLUDE statements identically:

```
%include payrec;
%include syslib(payrec);
```

For an example describing how to detect and exclude multiply included text, see the *PL/I Language Reference*.

%OPTION directive: The %OPTION directive is used to specify one of a selected subset of compile-time options for a segment of source code. The specified option is then in effect until one of the following occurs:

- Another %OPTION directive specifies a complementary compile-time option which overrides the first.
- A compile-time option saved using the %PUSH directive is restored using the %POP directive.

The compile-time options or directives that can be used with the %OPTION directive include:
- LANGLVL(SAA)
- LANGLVL(SAA2)

See Chapter 6, “Compile-time option descriptions,” on page 41 for option descriptions.

%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.

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 62). 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 "Specifying options in your source program" on page 37.

Program file format

The compiler, running under the AIX operating system, expects the contents of your source file to consist of ASCII format and LF type[1]. 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 LF).

Note: The compiler can interpret characters that are in the range X'00' to X'1F' 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). You should split long strings into a series of shorter strings (each of which fits on a line) that are concatenated together. For example, instead of coding this:

```
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;
```

You should use the following sequence of statements:

```
do;
 if x > 200 then
  display ('This is a long string and requires more than '
```

1. An LF type file is composed of lines of variable lengths, each delimited by the LF characters. LF is a special ASCII character that signifies "Line Feed"—hexadecimal value 0A. The compiler interprets LF as a record delimiter. The hexadecimal value 1A signifies the end of the file.

```
   ||'one line to type it into my program');
 else
  display ('This is a short string');
end;
```

Chapter 5. Compiling, linking, and running your program

Compiling, linking, and running your program

Invoking the PL/I for AIX compiler 29
Which invocation to use 29
How to specify options 29
Setting up the compilation environment . . . 30
Tailoring the configuration file 31
Input files 34
Specifying compile-time options 35
Specifying options in the configuration file . . 35
Specifying options on the command line. . . 35
Specifying options in your source program . . 37
Invoking the linkage editor 38
Running a program. 38

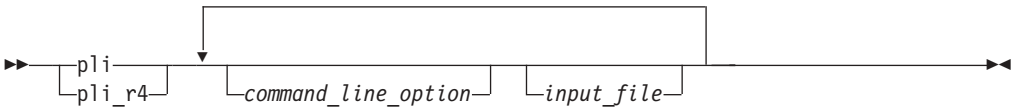
Once you have created a source file and understand portability implications, you prepare and run your program in four phases:

1. Preprocessing (optional, see Chapter 7, “PL/I preprocessors,” on page 83 for details)
2. Compiling
3. Linking
4. Running

This chapter discusses the details of using the PL/I for AIX compiler to invoke the compiler, specify available compile-time options, invoked the linkage editor, and run the executable program.

Invoking the PL/I for AIX compiler

To compile a source program, use either the the **pli** or **pli_r4** command.



Which invocation to use

pli

The **pli** command compiles PL/I source files, links the resulting object files with any object files and libraries specified on the command line in the order indicated, and produces a single executable file.

pli_r4

Use the **pli_r4** command to compile and link programs that access ENCINA files, including CICS ECI programs and PL/I batch programs.

How to specify options

You can specify options and input files in any order.

command_line_option

Specify a *command_line_option* in one of the following ways:

- **-q***option*
- Option flag, usually a single letter preceded by a minus sign(-).

If you choose to specify compile-time options on the command line, the format differs from either setting them in the configuration file or in your source file using %PROCESS statements.

For a list of possible options, see “Specifying compile-time options” on page 35.

input_file
Provide the AIX file specification for your program files. 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.

For a list of input files, see “Input files” on page 34.

Setting up the compilation environment

“Setting environment variables” on page 23 describes how environment variables are set for the AIX operating system. Instead of affecting the AIX environment, some environment variables more specifically affect the results of your compilation. Therefore, before using the compiler, you must set these environment variables and also set the attributes in your configuration file.

Message and help files: To use PL/I for AIX, you first need to install the message catalogs and help files and set the following environment variables:

LANG
Specifies the national language for message and help files.

NLSPATH
Specifies the path name of the message and help files.

The LANG environment variable can be set to any of the locales provided on the system. For more information on locales, refer to the AIX system reference manuals.

The national language code for United States English is `en_US`. If the appropriate message catalogs have been installed on your system, any other valid national language code can be substituted for `en_US`.

To determine the current setting of the national language on your system, use the following `echo` commands:

```
echo $LANG
echo $NLSPATH
```

These environment variables are initialized when the operating system is installed, and might differ from the ones you want to use.

You use different commands to set the environment variables depending on the shell you are using, see “Setting environment variables” on page 23. To set environment variables from the Bourne shell or Korn shell, use the following commands:

```
LANG=en_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%N

export LANG NLSPATH
```

To set the variables so that all users have access to them, add the commands to the file `/etc/profile`. To set them for a specific user only, add the commands to the file `.profile` in the user’s home directory. The environment variables are set each time the user logs in.

To set the environment variables from the C shell, use the following commands:

```
setenv LANG en_US
setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%N
```

In the C shell, you cannot set the environment variables so that all users have access to them. To set them for a specific user only, add the commands to the file `.cshrc` in the user's home directory. The environment variables are set each time the user logs in.

Defining library paths: PL/I for AIX requires that you specify `/usr/lpp/pli/lib` in the LIBPATH environment variable. The `/lib` and `/usr/lib` directories contain system libraries and should also be included in your LIBPATH directory list.

Locating online documentation: PL/I for AIX includes versions of the reference manuals that are viewable from your terminal. To let the system know where the documentation is located for use with the `xview` command, you must define the BOOKSHELF environment variable.

There are three directories that contain information:

```
/usr/lpp/pli/ipf  (PL/I for AIX documentation)
/usr/lpp/sde/ipf  (Program Builder and LPEX manuals)
/usr/lpp/SdU/ipf  (Smartdata Utilities documentation)
```

These three directories must, therefore, be set as follows:

```
export
  BOOKSHELF=/usr/lpp/pli/ipf:/usr/lpp/sde/ipf:/usr/lpp/SdU/ipf:$BOOKSHELF
```

Once the BOOKSHELF environment variable is set, you can view a list of all of the books available by entering the following command:

```
xview ibmpli.inf
```

You can access online help panels for the debug tool by pressing `F1` from within a Distributed Debugger session.

Tailoring the configuration file

The PL/I for AIX configuration file contains attributes that you can use to specify compile-time options or preprocessor option settings. The default configuration file is stored as `/etc/pli.cfg`.

You can copy this file to your user directory, give it a unique name *confignam*e, and make changes to the copy to support specific compilation requirements or to support other PL/I compilation environments. The **pli** command picks up the default file unless you use the **-F**<*confignam*e> flag to specify your own configuration file.

By adding stanzas to your copy of the configuration file, you can customize your own PL/I compilation environment. You can link the compiler invocation command to more than just the name of the configuration file, you can also specify which stanza of the configuration file the compiler uses.

To make links to select additional stanzas or to specify a stanza or another configuration file, you also use the **-F** option. Consider the following compiler invocation:

```
pli -Fmyconfig.cfg: SPECIAL myfile.pli
```

This would compile `myfile.pli` using the SPECIAL stanza in the configuration file that you created and named `myconfig.cfg`.

Configuration file attributes: The following table contains a list of attributes that could be found in a configuration file stanza:

Table 2. Configuration file attributes

Attribute	Description
crt	Path name of the object file passed as the first parameter to the linkage editor. If you do not specify either the **-p** or **-pg** option, the crt value is used. The default is /lib/crt.o.
gcrt	Path name of the object file passed as the first parameter to the linkage editor. If you specify the **-pg** option, the gcrt value is used. The default is /lib/gcrt.o.
ld	Path name to be used to link PL/I programs. The default is /bin/ld.
ldopt	List of options that are directed to the linkage editor part of the compiler. These override all normal processing by the compiler and are directed to the linkage editor.
libraries	Library options, separated by commas, that the compiler passes as the last parameters to the linkage editor. Specifies the libraries that the linkage editor is to use at link-edit time for both profiling and nonprofiling. The default is -lplishr,-lm,-lc.
mcrt	Path name of the object file passed as the first parameter to the linkage editor. If you specify the **-p** option, the *mcrt* value is used. The default is /lib/mcrt.o.
pliopt	A string of compile-time options to be processed by the compiler as if they had been entered on the %PROCESS The original defaults, together with the changes you apply using this attribute, become the new defaults. Any options you specify on the **pli** command or in your source program override these defaults.
sqlopt cicsopt macopt incopt	A string of options to be processed for the SQL preprocessor, CICS preprocessor, macro facility, or include preprocessor, respectively. These options are entered using the same format as if they were specified with a %PROCESS statement. The original preprocessor option defaults, along with the changes you apply using these attributes, become the new defaults. Any options you specify using the PP compile-time option (“PP” on page 68) with the **pliopt** attribute, on the command line, or in your source program override these options.
proflibs	Library options, separated by commas, that the compiler passes to the linkage editor when profiling options are specified. Specifies the profiling libraries used by the linkage editor at link-edit time. The default is -L/lib/profiled,-L/usr/lib/profiled.
use	Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the use stanza apply if no value is provided in the local, or default, stanza.

Sample configuration file: Figure 1 on page 33 shows a sample configuration file like the one you can find in /etc/pli.cfg.

```
*
* 5765-549 COMPONENT_NAME: PL/I for AIX Compiler
*
* FUNCTIONS: PL/I configuration file
*
* ORIGINS: 27
*
* (C) COPYRIGHT International Business Machines Corp. 1992,2004
* All Rights Reserved
* Licensed Materials - Property of IBM
*
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*

* standard PL/I compiler
pli:     use        = DEFLT
         crt        = /lib/crt0.o
         mcrt       = /lib/mcrt0.o
         gcrt       = /lib/gcrt0.o
         libraries  = -lplishr,-libmrtab,-lm,-lc
         proflibs   = -L/lib/profiled,-L/usr/lib/profiled
         pliopt     =
         sqlopt     =
         cicsopt    =
         macopt     =
         incopt     =

* standard PL/I compiler aliased as pli_r4 (DCE)
pli_r4:  use        = DEFLT
         crt        = /lib/crt0_r.o
         mcrt       = /lib/mcrt0_r.o
         gcrt       = /lib/gcrt0_r.o
         libraries  = <see list of libraries following figure>
         proflibs   = -L/lib/profiled,-L/usr/lib/profiled
         pliopt     =
         sqlopt     =
         cicsopt    =
         macopt     =
         incopt     =

* common definitions
DEFLT:  ld         = /bin/ld
        ldopt      = -T512 -H512 -bhalt:4
```

Figure 1. Sample configuration file

You must specify libraries on a single line in the configuration file. For `pli_r4`, specify the following libraries: -L/usr/lib/dce, -L/usr/lib/threads, -ldcelibc_r, -ldcepthreads, -lpthreads, -lplishr_r, -libmrtab, -lm, -lc_r, -lc

Creating your own configuration file: If you have a particular program that requires you to specify certain options each time you run it, you might find it useful to create your own copy of the configuration file. For example, if you have a particular application (or set of applications) that requires the use of the PL/I preprocessors, you could alter the default configuration file.

The following segment represents the attributes in the configuration file that control the compile-time options and preprocessor options:

```
pliopt     = opt(2) pp(macro)
sqlopt     = source
cicsopt    = source
macopt     = fixed(bin)
incopt     = -inc
```

Setting the options as illustrated in this example would have the following effect:

- Your compilation would be optimized.
- The macro facility would be invoked.
- The macro facility suboption of FIXED(BIN) would be in effect (instead of the default FIXED(DEC)).
- SOURCE would be in effect if you invoked the CICS and SQL preprocessors.
- All lines that start with `-inc` would be treated as include directives if you invoked the include preprocessor either on the command line or in a %PROCESS statement.

You could save this configuration file as `my.cfg` in the `/usr/mydir/pli` directory. In order to use your configuration file instead of the system configuration file, you would have to specify **-F** (see "Single and multiletter flags" on page 36) and the name of your configuration file on the command line. Either of the following examples would have the same effect as you compile and link the program `myprog.pli`: **Example 1**

```
pli -qoptions, pp=sql:cics -Fmy.cfg myprog.pli
```

Example 2

```
pli -qoptions, pp=sql:cics -F/usr/mydir/pli/my.cfg myprog.pli
```

Either of these examples would cause the compiler to invoke the macro facility and CICS and SQL preprocessors. The suboptions set in `my.cfg` would be in effect for the compilation.

If, however, you specified the following command:

```
pli -qpp=macro=fixed=dec -Fmy.cfg
```

The FIXED(DEC) option on the command line would override the FIXED(BIN) option set in the `macopt` attribute of `my.cfg`.

The options set in the configuration file override the defaults set by the compiler. However, you can override the defaults in the configuration file by specifying options on the command line or in a %PROCESS statement.

If you choose to specify options in the `/etc/pli.cfg` configuration file, remember that these options are in effect for every compilation unless you override them. If you invoke the macro facility in the system configuration file, for example, you could negatively affect the rate of compilation for programs that do not require macro processing.

Input files

The **pli** command accepts the following types of files:

Source files—.pli

All `.pli` files are source files for compilation. The **pli** command sends source files to the compiler in the order they are listed. If the compiler cannot find a specified source file, it produces an error message and the **pli** command proceeds to the next file if one exists.

Object files—.o

All `.o` files are object files. The **pli** command sends all object files along with library files to the linkage editor (`ld`) at link-edit time unless you specify the **-c** option. After it compiles all the source files, the compiler invokes the linkage editor to link-edit the resulting object files with any object files specified in the input file list, and produces a single executable output file.

Library files—.a

The **pli** command sends all of the library files (`.a` files) to the linkage editor at link-edit time.

Specifying compile-time options

PL/I for AIX provides compile-time options to change any of the compiler's default settings. You can specify options on the command line, and they remain in effect for all compilation units in the file, unless %PROCESS statements in your source program override them. Any options that you can specify on the command line can also appear in the configuration file, and remain in effect for all compilation units unless the command line or %PROCESS statement override them.

Refer to Chapter 6, "Compile-time option descriptions," on page 41 for a description of these options.

Specifying options in the configuration file

Options set in the configuration file override the default settings. They can be overridden by options set on the command line or in the source file.

The configuration file is described in "Tailoring the configuration file" on page 31. You can use the **pliopt** attribute of the configuration file to set compile-time options instead of retyping them on the command line each time you invoke the compiler. You can also use **sqlopt**, **cicsopt**, **macopt**, and **incopt** attributes to set preprocessor compile-time options in the configuration file.

The format for compile-time options specified in the configuration file is depicted by the syntax diagrams. Options set on the command line are different and must be preceded by **-q**. Each compile-time option in Chapter 6, "Compile-time option descriptions," on page 41 has a section showing the **Command line syntax** in addition to the syntax diagram.

As an example, suppose you want to specify the OPTIMIZE compile-time option ("OPTIMIZE" on page 67). To set this option in the configuration file (either in the default file or in your own copy), you would specify the following:

```
pliopt = opt(2)
```

To set this same option on the command line, however, you would enter:

```
-qopt=2
```

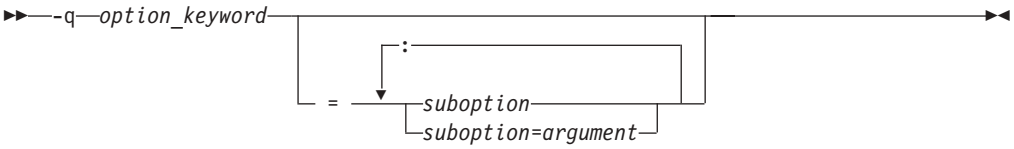
Specifying options on the command line

Command line options override the default settings of the option and any set in the configuration file. They are overridden by options set in the source file.

You can specify compile-time options on the command line in two ways:

- **-q***option_keyword* (compiler-specific)
- Single and multiletter flags

-qoption_keyword: You can specify options on the command line using the **-q***option* format.



You can have multiple **-q***options* on the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the **-q** in lowercase.

Some compile-time options allow you to specify suboptions. These suboptions are indicated on the command line with an equal sign following the **-q***option_keyword*. Multiple suboptions must be separated with a colon(:) and no intervening blanks.

An option, for example, that contains multiple suboptions is RULES ("RULES" on page 72). To specify RULES on the command line with two suboptions, you would enter:

```
-qrules=ibm:laxdcl
```

The LIMITS option ("LIMITS" on page 60) is slightly more complex since each of its suboptions also has an argument. You would specify LIMITS on the command line as shown in the following example:

```
-qlimits=extname=31: fixeddec=15
```

To specify these options in the configuration file, you would use the 370-style notation as illustrated their respective syntax diagrams:

```
pliopt = rules(ibm,laxdcl) limits(extname(31) fixeddec(15))
```

Single and multiletter flags: The XL family of compilers uses a number of common conventional flags. Each language has its own set of additional flags.

Some flag options have arguments that form part of the flag, for example:

```
pli samp.pli -F/home/tools/test3/new.cfg: mypli -qgonumber
```

In this case, `new.cfg` is a custom configuration file.

You can specify flags that do not take arguments in one string:

```
pli -Ogc samp1.pli
```

Specifying the flags in one string has the same effect as specifying the same options separately.

```
pli -O -g -c samp1.pli
```

Both examples compile the PL/I source file `samp1.pli` with optimization (**-O**) and produce symbolic information used by the debugger (**-g**), but do not invoke the linkage editor (**-c**).

You can specify one flag option that takes arguments as part of a single string, but it must be the last option specified. For example, you can use the **-F** flag (to specify the name of an alternate configuration file) together with the other flags, only if the **-F** flag and its argument are specified last:

```
pli -OgFmy.cfg samp1.pli
```

The string of flags in the preceding example is equivalent to the following:

```
pli -O -g -Fmy.cfg samp1.pli
```

Most flag options are a single letter, but some are two letters. Specifying **-pg** (extended profiling) is not the same as **-p -g** (profiling and generating debug information). Do not specify two or more options in a single string if there is another flag that uses that letter combination.

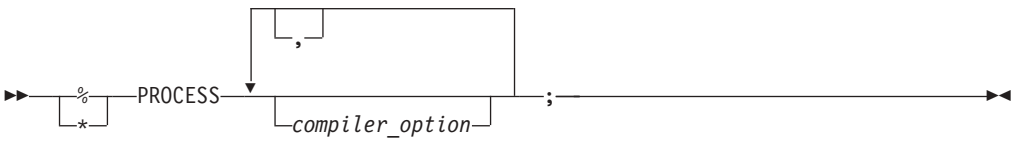
Table 3. Compile-time option flags supported by PL/I for AIX

Option	Description
-c	Compile only.
-F<*file*>*	Names an alternate configuration file.
-g	Produce symbolic information used by the debugger. This option is equivalent to -qGN.
-I<*dir*>*	Add path <*dir*> to the directories to be searched for INCLUDE files. -I must be followed by a path and only a single path is allowed per -I option. To add multiple paths, use multiple -I options. There shouldn't be any spaces between -I and the path name.
-O, -O2	Optimize generated code. This option is equivalent to -qOPT=2.
-q<*option*>*	Pass it to the compiler. <*option*> is a compile-time option. Each option should be delimited by a comma and each suboption should be delimited by an equal sign or colon. There shouldn't be any spaces between **-q** and <*option*>.
-p	Set up the object files produced by the compiler for profiling.
-pg	Set up the object files for profiling, but provides more information than is provided by the -p option.
-v	Display compile and link steps and execute them.
-#	Display compile and link steps, but do not execute them.
Note: *You must specify an argument where indicated; otherwise, the results are unpredictable.	

Specifying options in your source program

Compile-time options set in the source file override default settings, options set in the configuration file, options set on the command line. Options set in the source program, however, apply only to the current compilation.

To specify options in your source file, use the %PROCESS (or *PROCESS) statement.



The following example illustrates the use of the %PROCESS statement:

```
% 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.

Invoking the linkage editor

The **ld** command (the linkage editor or binder) combines the designated object files, import lists and libraries into one object file, resolving external references.

The **pli** command both compiles and links your files. If you specify the **-c** flag., however, PL/I for AIX only compiles the source program and creates an object file. In this case, you can link edit by issuing the **pli** command a second time (without **-c**), or using the **ld** command. The order of the object files determines what symbol the linker accepts.

If you use **ld** or **pli** to invoke the linkage editor for a fetchable routine, you must specify **-e**<*name*>, where <*name*> is the name of the entry point into the fetchable routine.

Table 4. Most frequently used linking options

Option	Description

-b<*option*>*	Pass a binder option to the linker. Some binder options include:
	`-bhalt:num  Specifies the maximum error level for binder command processing to continue.`
	`-bloadmap:name  Requests that a log of linker actions and messages be saved in file name.`

-l<*key*>*	Search the specified library file.
-L<*dir*>*	Search in directory <*dir*> for files specified by -l<*key*>.
-o<*name*>*	Provides a name for the executable file. If not specified, the name of the executable defaults to the file name of the `.pli`. If multiple `.pli` files are used, the executable defaults to the name of the first file specified.
Note: *You must specify an argument where indicated; otherwise, the results are unpredictable.	

For more information on the linkage editor, see **ld** in the AIX Commands Reference.

Running a program

You can run a program by entering the path name and file name of an executable file. The default file name for the executable file is the name of the `.pli` source file

(with no extension). You can select a different name by using the **-o** flag. If you specified **-o***name*, the output filename is *name*. You should try to use unique names for your executable files to avoid confusion. If, for example, *name* is the same as a system command and you do not specify the absolute path name of the executable program, you could invoke the system command instead of your program.

Chapter 6. Compile-time option descriptions

Compile-time option descriptions 41
Rules for using compile-time options 43
AGGREGATE 44
ATTRIBUTES 44
BIFPREC 45
BLANK. 46
CHECK 46
CMPAT. 46
CODEPAGE 47
COMPILE 47
COPYRIGHT 48
CURRENCY 48
DEFAULT 48
EXIT. 54
EXTRN 54
FLAG 55
FLOATINMATH. 55
GONUMBER 56
GRAPHIC 56
IMPRECISE 56
INCAFTER 57
INITAUTO 57
INITBASED 57
INITCTL 58
INITSTATIC 58
INCDIR 58
INSOURCE 58
LANGLVL. 59
LIMITS 60
LINECOUNT. 60
LIST. 61
MACRO 61
MARGINI 61
MARGINS. 62
MAXMEM. 62
MAXMSG 63
MAXSTMT 64
MAXTEMP 64
MDECK 64
MSG. 64
NAMES 65
NATLANG 65
NEST 65
NOT. 66
NUMBER 66
OPTIMIZE. 67
OPTIONS 67
OR 67
PP 68
PPTRACE 69
PRECTYPE 69
PREFIX. 69
PROCEED. 70
REDUCE 71
RESEXP 71
RESPECT 72
RULES 72
SEMANTIC 75
SOURCE 76
STATIC. 76
SPILL 76
STMT 77
STORAGE. 77
SYNTAX 77
SYSPARM 78
SYSTEM 78
TERMINAL 78
TEST 79
USAGE. 79
WIDECHAR 79
WINDOW. 80
XINFO 80
XREF 81

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 5 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 5 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 111.

Table 5. Compile-time options, abbreviations, and IBM-supplied defaults

Compile-Time Option	Abbreviated Name	AIX Default
AGGREGATE (DECIMAL | HEXADEC)| NOAGGREGATE	AG | NAG	NOAGGREGATE
ATTRIBUTES[(FULL|SHORT)] | NOATTRIBUTES	A | NA	NA [(FULL)][1]
BIFPREC(15 | 31)	–	BIFPREC(31)
BLANK('c')	–	BLANK('t')[2]
CHECK(STORAGE | NOSTORAGE)	–	CHECK(NSTG)
CMPAT(LE | V1 | V2)	–	CMPAT(V2)
CODEPAGE(n)	CP	CODEPAGE(00819)
COMPILE | NOCOMPILE[(W | E | S)]	C | NC	NOCOMPILE(S)
COPYRIGHT('string') | NOCOPYRIGHT	–	NOCOPYRIGHT
CURRENCY('c')	CURR	CURRENCY($)
DEFAULT(*attribute* | *option*)	DFT	See page 48
EXIT | NOEXIT	–	NOEXIT
EXTRN(FULL | SHORT)	–	EXTRN(FULL)
FLAG[(I | W | E | S)]	F	FLAG(W)
FLOATINMATH(ASIS | LONG | EXTENDED)	–	FLOATINMATH(ASIS)
GONUMBER | NOGONUMBER	GN | NGN	NOGONUMBER
GRAPHIC | NOGRAPHIC	GR | NGR	NOGRAPHIC
IMPRECISE | NOIMPRECISE	–	IMPRECISE
INCAFTER([PROCESS(*filename*)])	–	INCAFTER()
INCDIR('*directory name*')	–	INCDIR()
INCLUDE[(EXT('*include extension*'))]	INC	INC(EXT('inc'))
INITAUTO | NOINITAUTO	–	NOINITAUTO
INITBASED | NOINITBASED	–	NOINITBASED
INITCTL | NOINITCTL	–	NOINITCTL
INITSTATIC | NOINITSTATIC	–	NOINITSTATIC
INSOURCE[(FULL|SHORT)] | NOINSOURCE	IS | NIS	NOINSOURCE
INTERRUPT | NOINTERRUPT	INT | NINT	NOINTERRUPT
LANGLVL(SAA | SAA2[,NOEXT | OS])	–	LANGLVL(SAA2,OS)
LIMITS(*options*)	–	See page 60
LINECOUNT(*n*)	LC	LINECOUNT(60)
LIST | NOLIST	–	NOLIST
MACRO | NOMACRO	M | NM	NOMACRO
MARGINI('c') | NOMARGINI	MI | NMI	NOMARGINI
MARGINS(m,n[,c])| NOMARGINS	MAR(m,n)	MARGINS F-format: (2,72) V-format: (10,100)
MAXMEM(n)	–	MAXMEM(2048)
MAXMSG(I | W | E | S,n)	–	MAXMSG(W,250)

Table 5. Compile-time options, abbreviations, and IBM-supplied defaults (continued)

Compile-Time Option	Abbreviated Name	AIX Default
MAXSTMT(n)	–	MAXSTMT(4096)
MAXTEMP(n)	–	MAXTEMP(1000)
MDECK | NOMDECK	MD | NMD	NOMDECK
NAMES('*lower*'[,*upper*])	–	NAMES('#@$','#@$')
NATLANG(ENU | UEN)	–	NATLANG(ENU)
NEST | NONEST	–	NONEST
NOT	–	NOT('¬')
NUMBER | NONUMBER	NUM | NNUM	NUMBER
OPTIMIZE(0 | 2) | NOOPTIMIZE	OPT | NOPT	OPT(0)
OPTIONS | NOOPTIONS	OP | NOP	NOOPTIONS
OR('c')	–	OR(' | ')
PP(*pp-name*) | NOPP	–	NOPP
PPTRACE | NOPPTRACE	–	NOPPTRACE
PRECTYPE (ANS | DECDIGIT | DECRESULT)	–	PRECTYPE(ANS)
PREFIX(*condition*)	–	See page 69
PROCEED | NOPROCEED[(W | E | S)]	PRO | NPRO	NOPROCEED(S)
REDUCE | NOREDUCE	–	REDUCE
RESEXP | NORESEXP	–	RESEXP
RESPECT([DATE])	–	RESPECT()
RULES(*options*)	LAXCOM | NOLAXCOM	See page 72
SEMANTIC | NOSEMANTIC[(W | E | S)]	SEM | NSEM	NOSEMANTIC(S)
SOURCE | NOSOURCE	S | NS	NOSOURCE
STATIC(FULL | SHORT)	–	STATIC(SHORT)
STMT | NOSTMT	–	NOSTMT
STORAGE | NOSTORAGE	STG | NSTG	NOSTORAGE
SYNTAX | NOSYNTAX[(W | E | S)]	SYN | NSYN	NOSYNTAX(S)
SYSPARM('string')	–	SYSPARM('')
SYSTEM(AIX | CICS)	–	SYSTEM(AIX)
TERMINAL | NOTERMINAL	TERM | NTERM	
TEST | NOTEST	–	NOTEST[3]
USAGE(*options*)	–	See page 79
WIDECHAR(BIGENDIAN | LITTLEENDIAN)	WCHAR	WIDECHAR(BIGENDIAN)
WINDOW(w)	–	WINDOW(1950)
XINFO(*options*)	–	XINFO(NODEF,NOXML)
XREF[(FULL | SHORT)] | NOXREF	X | NX	NX [(FULL)][1]

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

The syntax diagram for each compile-time option shows how you would specify the option either in a PROCESS statement or in the configuration file. The command line syntax for each option is slightly different than shown in the syntax diagram. Each option, therefore, also has at least one example of how you could specify that option on the AIX command line.

Rules for using compile-time options

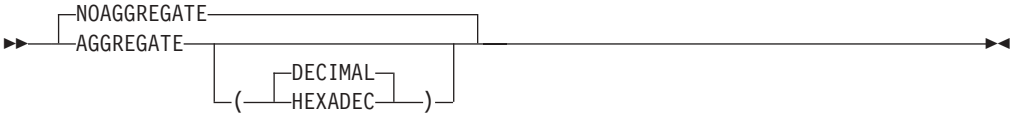
1. If you specify mutually exclusive compile-time options or suboptions, the last one you specify takes effect.
2. If required strings conform to PL/I identifier rules, you do not need to enclose them in quotes. The compiler folds these strings to uppercase.

The following options should have their string specifications enclosed in quotes, 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 quotes, the string itself cannot contain any quotes.
4. If an option has a string enclosed in quotes, the string can be specified as a hex string, for example NOT('aa'x).
5. If you incorrectly specify any compile-time options—for example, if you specify NEXT instead of NEST—the OPTIONS compile-time option is automatically set to OPTIONS. This provides you with a listing of all compile-time options in effect for the compilation.

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

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

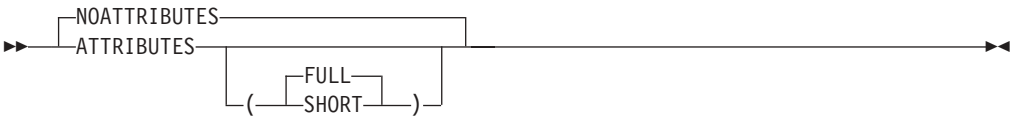
HEXADEC
All offsets in the aggregate listing will be displayed in hexadecimal.

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 111.

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 111.

SHORT
Omit unreferenced identifiers.

BIFPREC

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

For best compatibility with PL/I for MVS & VM, OS PL/I V2R3 and earlier compilers, 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:

```
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 four-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.

BLANK

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

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)

CHECK

This option causes the compiler to monitor ALLOCATE and FREE statments.

ABBREVIATIONS: STG, NSTG

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

CMPAT

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

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.

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

The DFT(DESCLIST) option conflicts with the CMPAT(V1) or CMPAT(V2) option, and if it is specified with either the CMPAT(V1) or the CMPAT(V2) option, a message will be issued and the DFT(DESCLOCATOR) option assumed.

CODEPAGE

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

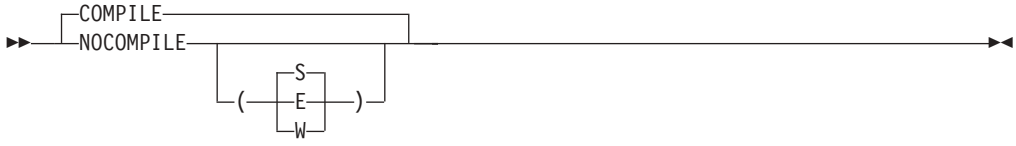
The supported CCSID's are:

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

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

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.

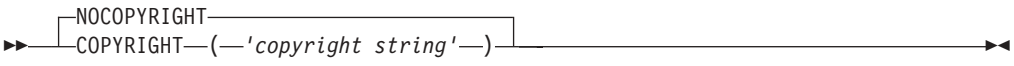
NOCOMPILE(W)
Compilation halts if a warning, error, severe error, or unrecoverable error is detected.

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.

COPYRIGHT

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

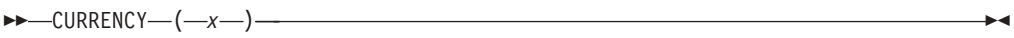


The string is limited to 1000 characters in length.

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

CURRENCY

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

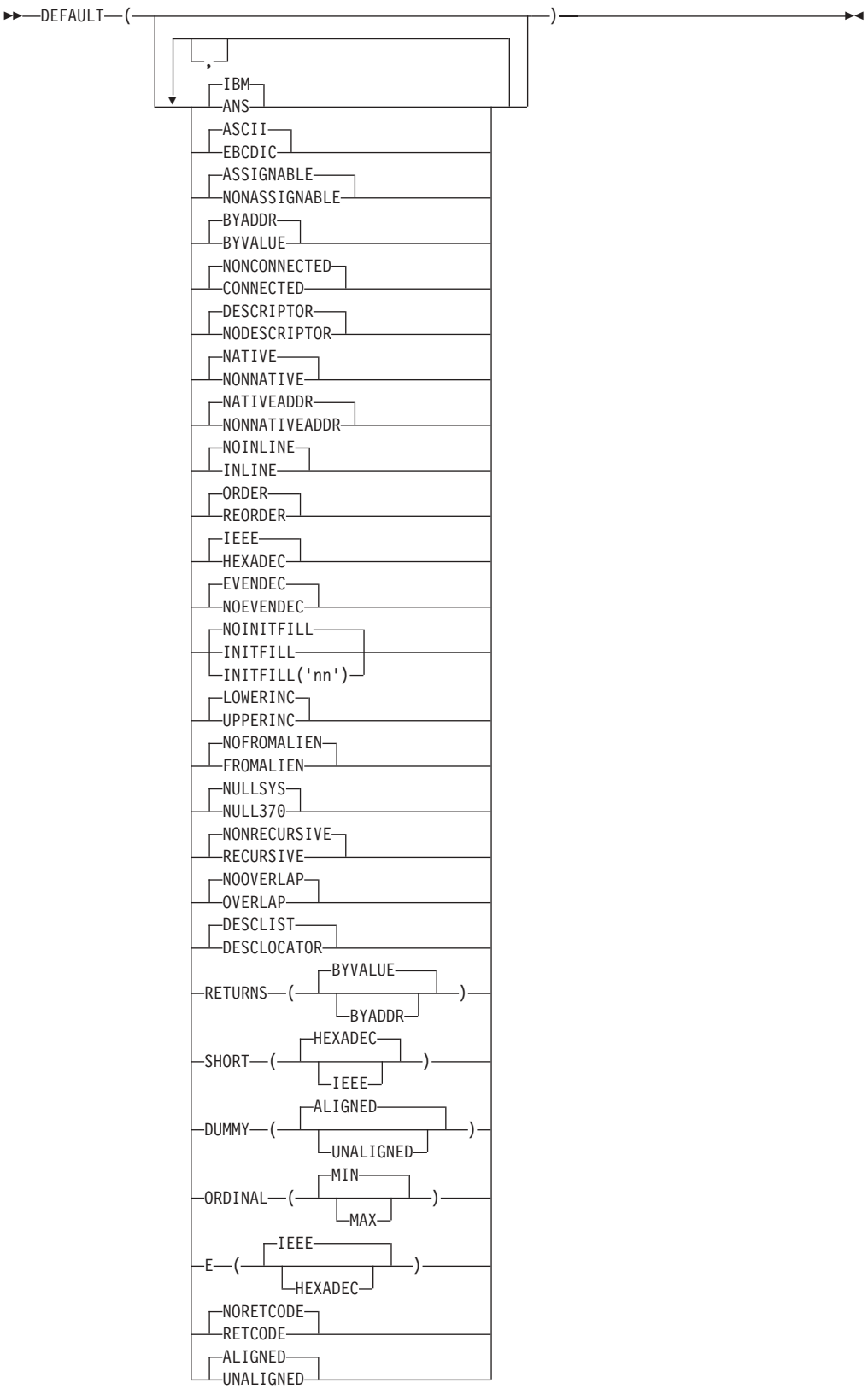


x Character that you want the compiler and runtime to recognize and accept as the dollar sign in picture strings.

DEFAULT: CURRENCY('$')

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.



ABBREVIATIONS: DFT, ASGN, NONASGN, CONN, NONCONN

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

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

Under the IBM suboption, 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).

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.

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
- 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.

NATIVE or NONNATIVE

This option affects only the internal representation of fixed binary, ordinal, 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.

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 15, “Improving performance,” on page 237.

ORDER or REORDER

Affects optimization of the source code. Specifying REORDER allows optimization of your source code, see Chapter 15, “Improving performance,” on page 237.

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.

IEEE or HEXADEC

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

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 of you were using mainframe PL/I).

EVENDEC is the default.

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

If you specify INITFILL with a hex value (nn), that value is replicated and fills storage for all automatic variables. If you do not enter a hex value, the default is '00'x. NOINITFILL does no initialization of these variables. INITIFILL 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.

NOINITIFILL is the default.

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.

NOFROMALIEN or FROMALIEN
If you specify NOFROMALIEN, options FROMALIEN is not assumed unless it is explicitly specified in your code. If you specify the FROMALIEN suboption, however, the compiler assumes options FROMALIEN on all external procedures encountered during compilation.

NOFROMALIEN 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.

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.

DESCLIST or DESCLOCATOR
When you specify DEFAULT(DESCLIST), 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(DESCLOCATOR), 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.

DESCLIST is the default.

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.

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:

```
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.

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.

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.

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.

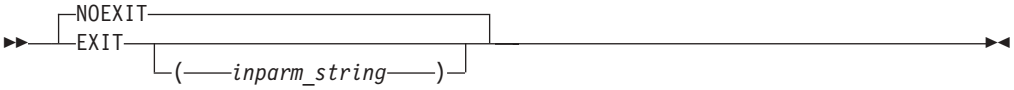
DFT(E(HEXADEC)) is useful in developing and testing 390 applications on the workstation. The statement ″put skip edit(x) (e(15,8));″ would produce no messages on 390, but, by default, it would be flagged under Intel and AIX. Specifying DFT(E(HEXADEC)) would fix this.

IEEE is the default.

DEFAULT (IBM ASCII ASSIGNABLE BYADDR NONCONNECTED DESCRIPTOR NATIVE NOINLINE ORDER IEEE EVENDEC NOINITFILL UPPERINC NOFROMALIEN ORDINAL(MIN) NOOVERLAP NULLSYS NONRECURSIVE DESCLIST RETURNS(BYVALUE) SHORT(HEXADEC) DUMMY(ALIGNED) NORETCODE ALIGNED E(IEEE)).

EXIT

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



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

For more information, see Chapter 14, “Using user exits,” on page 229.

EXTRN

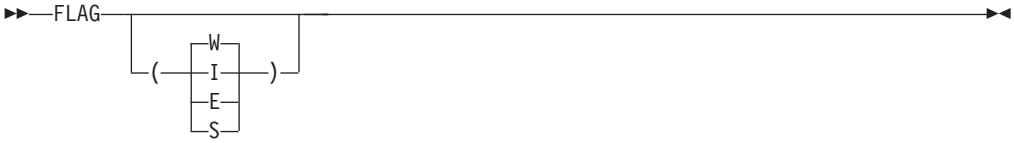
The EXTRN option controls when EXTRNs are emitted for external entry constants.

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.

FLAG

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.

FLOATINMATH

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

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

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

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

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

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

GONUMBER

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

ABBREVIATIONS: NGN, GN

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

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.

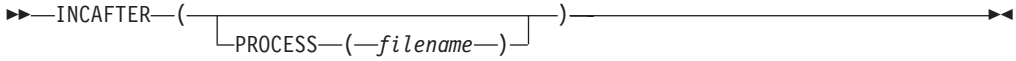
Use of this option produces smaller object code that runs faster. It is recommended for your production programs.

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.

INCAFTER

This option allows you to specify a file to be included after a particular statement in your source program.



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

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

Consider the following example:

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

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

INITAUTO

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:

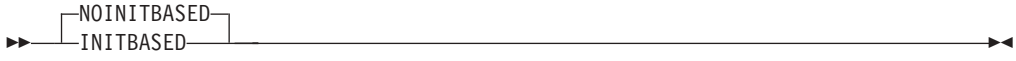
- INIT((*) 0) if it is FIXED or FLOAT
- INIT((*) '') if it is PICTURE, CHAR, BIT, GRAPHIC or WIDECHAR
- INIT((*) sysnull()) if it is POINTER or OFFSET

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.

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.

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.

INCDIR

This option allows you to include a directory in the search path for the location of include files.

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

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

INSOURCE

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.

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.

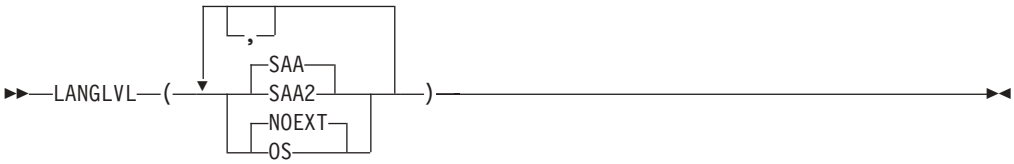
```
insource: proc options(main);
  %include member;
end;
```

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).

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.



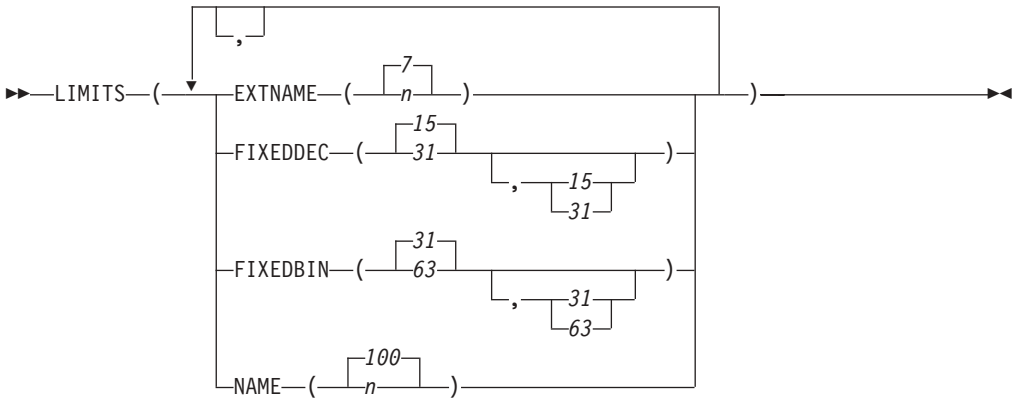
SAA
The compiler flags keywords that are not supported by OS PL/I Version 2 Release 3 and does not recognize any built-in functions not supported by OS PL/I Version 2 Release 3.

SAA2
The compiler accepts the PL/I language definition contained in the *PL/I Language Reference*.

NOEXT
No extensions beyond the language level specified are allowed.

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. Under FIXEDDEC(15,31), the attribute of an operation will have a precision greater than 15 only if one of the operands does.

The default is FIXEDDEC(15,31).

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 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) 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)

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 111.

MACRO

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 68.

MARGINI

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.

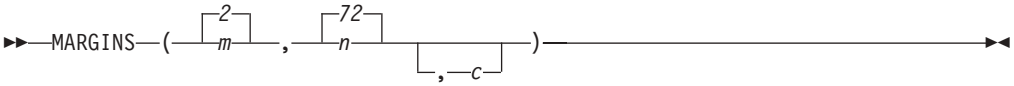
DEFAULT: MARGINI(' ')

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 111.

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.



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 ANS 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 ANS control character is present. Only the following control characters can be used:

(blank)
Skip one line before printing

0 Skip two lines before printing

– Skip three lines before printing

+ No skip before printing

1 Start new page

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

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

Specifying MARGINS(,,c) is an alternative to using %PAGE and %SKIP statements (described in *OS PL/I Version 2 Language Reference*).

DEFAULT: MARGINS (2 72)

MAXMEM

This option limits the amount of memory used for local tables of specific, memory-intensive optimizations.

size
The value, in kilobytes, of the memory limit used for local tables of specific, memory-intensive optimizations. If that memory is insufficient for a particular optimization, the quality of the optimization is reduced.

DEFAULT: MAXMEM(2048)

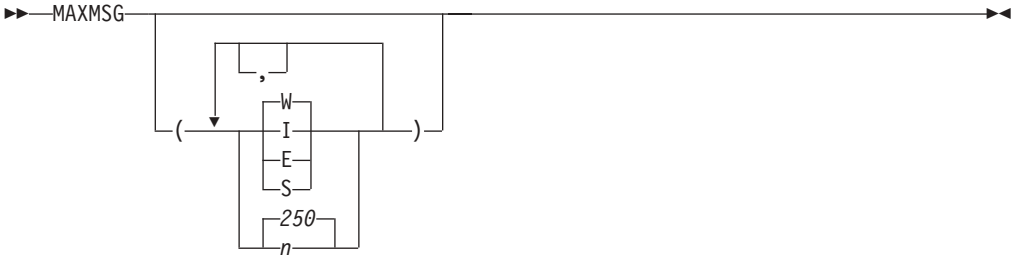
Notes:

1. A value of -1 permits each optimization to take as much memory as it needs without checking for limits. Depending in the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this might exceed available system resources.
2. The limit set by MAXMEM is the amount of memory for specific optimization phases, and not for the compiler as a whole. Tables required during the entire compilation process are not affected or include in this limit.
3. Setting a large limit has no negative effect on the compilation of source files where the compiler needs less memory.
4. Reduced optimization does not necessarily mean that the resulting program will be slower, only that the compiler cannot finish looking for opportunities to increase performance.
5. Increasing the limit does not necessarily mean that the resulting program is faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Depending on the source file being compiled, the size of the subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high might lead to page space exhaustion. In particular, specifying MAXMEM(-1) allows the compiler to try and use an infinite amount of storage, which in the worst case can exhaust the resources of even the most well-equipped machine.

MAXMSG

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



I Count all messages.

W Count all except information messages.

E Count all except warning and information messages.

S Count only severe error and unrecoverable error messages.

n Terminate the compilation if the number of messages exceeds this value. If messages are below the specified severity or are filtered out by a compiler exit routine, they are not counted in the number. 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)

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.

DEFAULT: MAXSTMT(4096)

MAXTEMP

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

max
The limit for the number of bytes that can be used for compiler-generated temporaries. The compiler flags any statement that uses more bytes than 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.

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 61 for an example.

MSG

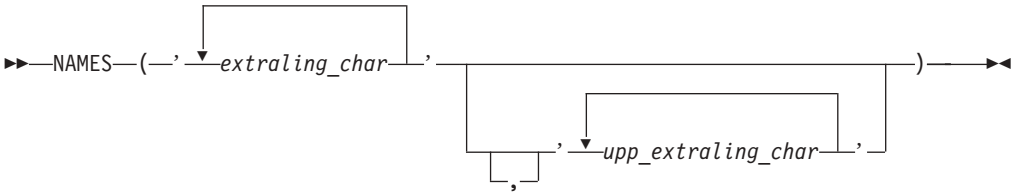
This option controls when the compiler will issue messages for conversions that will be done via a library call.

***** 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.

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*.



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('#@$' '#@$')

Command line example:

```
-qnames='äöüß'
```

NATLANG

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

ENU
US English, mixed case.

UEN
US English, upper case.

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 111.

NOT

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

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.

Command line example:

```
-qnot="^"
```

NUMBER

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

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 compiler turns off optimization if it detects the use of certain language constructs, for example out-of-block GOTOs.

See Chapter 15, “Improving performance,” on page 237 for a full discussion of optimization.

OPTIONS

This option produces a listing of all compile-time options in effect for the compilation (see “Compilation output” on page 111 for an example).

ABBREVIATIONS: NOP, OP

OR

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).

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:

```
or('\}')
or('|\}')
```

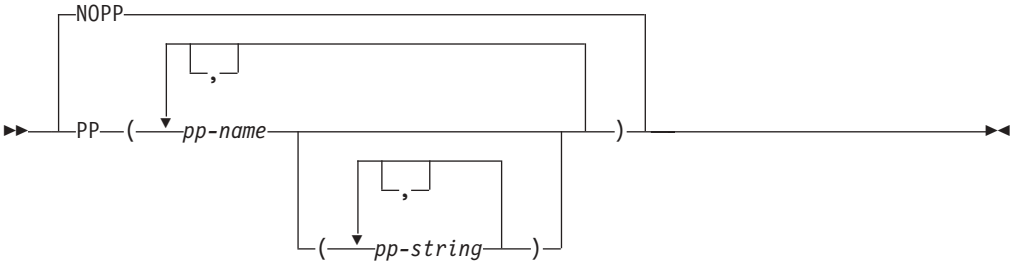
If you are invoking the compiler and specifying any compile-time options that use vertical bars (|) or a caret (^) on the command line, use double quotes around the character.

Command line example:

```
-qnot="^"
-qor="|"
```

PP

This option specifies which (and in what order) preprocessors are invoked prior to compilation. 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. The same preprocessor can be specified multiple times.



pp-name
The name given to a particular preprocessor. INCLUDE, MACRO, SQL, and CICS are the defined names for the preprocessors presently available. Using an undefined name causes a diagnostic error.

pp-string
A string of up to 100 characters representing the options for the corresponding preprocessor. If more than one *pp-string* is specified, they are concatenated with a blank separating each string.

Preprocessor options are processed from left to right. If two conflicting options are used, the last one specified is used.

You can set the default options for the `pp-string` for each of the preprocessors by using the appropriate attributes in the configuration file. See “Configuration file attributes” on page 32.

DEFAULT: NOPP

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.

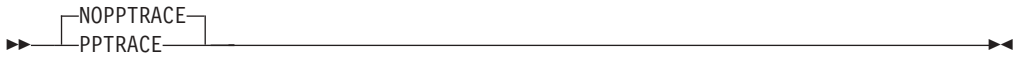
```
pp(macro('x') sql('dbname(sample)') macro)
```

Command line example:

```
-qpp=macro='x':sql='dbname=sample':macro
-qpp=sql='dbname=sample'
-qpp=cics=noedf:nodebug:macro:sql
```

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.

PRECTYPE

The PRECTYPE option determines how the compiler derives the attributes for the MULTIPLY, DIVIDE, ADD and SUBTRACT built-in functions when the operands are FIXED BIN and only a precision has been specified.

The PRECTYPE option has three suboptions:

ANS
Under PRECTYPE(ANS), the value p in BIF(x,y,p) 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).

DECDIGIT
Under PRECTYPE(DECDIGIT), the value *p* in BIF(x,y,p) is interpreted as specifying a decimal number of digits, the operation is performed as a binary operation and the result has the attributes FIXED BIN(s) where *s* is the corresponding binary equivalent to *p* (namely s = ceil(3.32*p)).

DECRESULT
Under PRECTYPE(DECRESULT), the value *p* in BIF(x,y,p) is interpreted, as also true for DECDIGIT, as specifying a decimal number of digits, but the operation is performed as a decimal operation and the result has the attributes FIXED DEC(p,0).

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*.

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

Examples:

Given the following source:

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

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

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

Command line example:

```
-qprefix=noconversion:nofixedoverflow
-qprefix=noinvalidop
```

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.

REDUCE

The REDUCE option specifies that the compiler is permitted to reduce an assignment of a null string to a structure into a simple copy operation - even if that means padding bytes might be overwritten.

The 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.

The REDUCE option will cause fewer lines of code to be generated for an assignment of a null string to a structure, and that will usually mean your compilation will be quicker and your code will run much faster. However, padding bytes may be zeroed out.

For instance, in the following structure, there is one byte of padding between *field11* and *field12*.

```
dcl
 1 sample ext,
   5  field10          bin fixed(31),
   5  field11          dec fixed(13),
   5  field12          bin fixed(31),
   5  field13          bin fixed(31),
   5  field14          bit(32),
   5  field15          bin fixed(31),
   5  field16          bit(32),
   5  field17          bin fixed(31);
```

Now consider the assignment *sample = '';*

Under the NOREDUCE option, it will cause eight assignments to be generated, but the padding byte will be unchanged.

However, under REDUCE, the assigment would be reduced to three operations.

RESEXP

The RESEXP option specifies that the compiler is permitted to evaluate all restricted expressions at compile time even if this would cause a condition to be raised and the compilation to end with S-level messages.

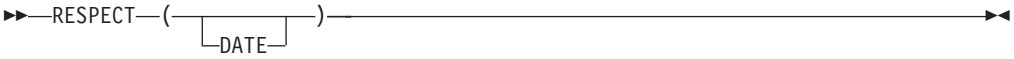
Under the NORESEXP compiler option, the compiler will still evaluate all restricted expression occurring in declarations, including those in INITIAL value clauses.

For example, under the NORESEXP option, the compiler would not flag the following statement (and the ZERODIVIDE exception would be raised at run-time)

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

RESPECT

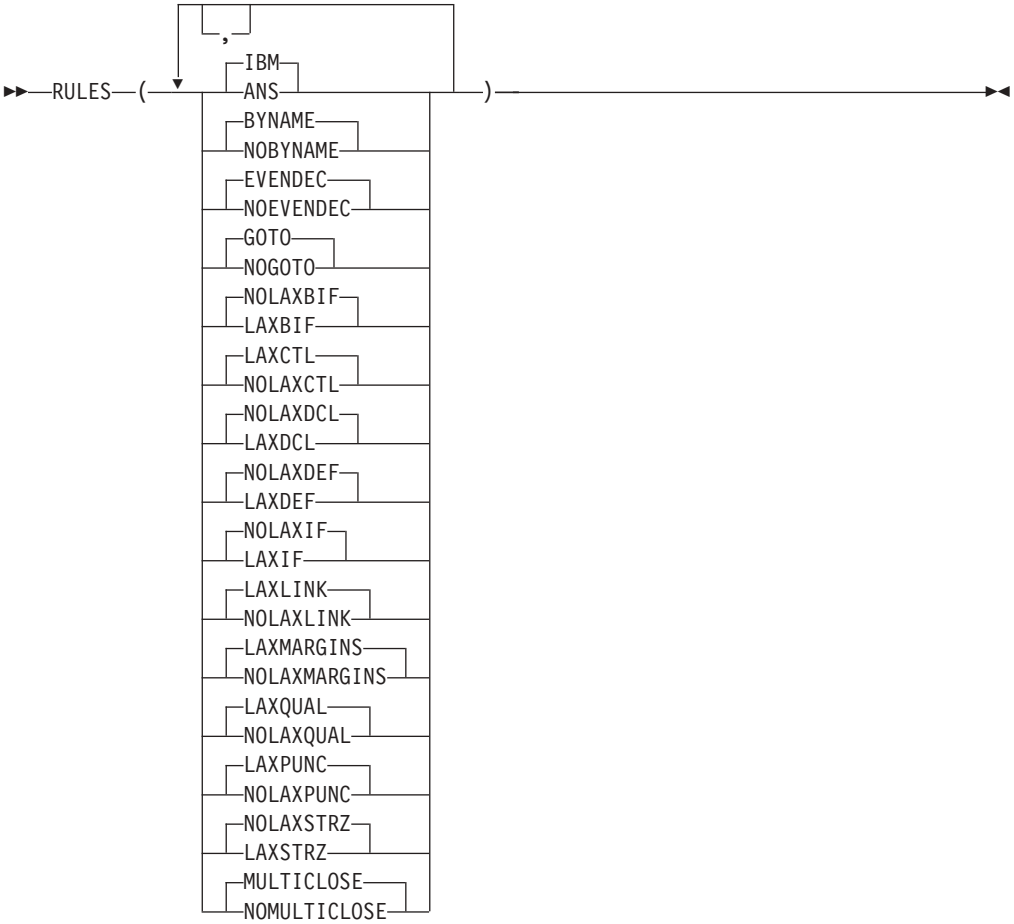
Causes the compiler to honor any specification of the DATE attribute and to apply the DATE attribute to the result of the DATE built-in function.



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.



ABBREVIATIONS: LAXCOM, NOLAXCOM

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.

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.

EVENDEC or NOEVENDEC

Specifying NOEVENDEC causes the compiler to flag all FIXED DECIMAL variables with an even precision with an E-level message.

GOTO or NOGOTO

GOTO causes the compiler to ignore all GOTO statements. GOTO is the default.

Use NOGOTO to have all GOTO statements flagged.

LAXBIF or NOLAXBIF

LAXBIF causes the compiler to build contextual declares for all built-in functions, including built-in functions that do not have an argument list.

NOLAXBIF is the default.

LAXCOMMENT or NOLAXCOMMENT

If you specify RULES(LAXCOMMENT), the compiler ignores the special characters /*/. Whatever comes between sets of these characters, then, is interpreted as part of the syntax rather than as a comment. If you specify

RULES(NOLAXCOMMENT), the compiler treats /*/ as the start of a comment which continues until a closing */ is found.

The SQL preprocessor objects to the DATE attribute. However, if you enclose the attribute between /*/ and /*/, the SQL preprocessor ignores it (as part of a comment that stretches from the first /* to the last */).

Enclosing the date attribute as described allows the host compiler to accept it as well.

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 is illegal under NOLAXCTL:

```
dcl a bit(8) ctl;
alloc a 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 | 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).

LAXIF or NOLAXIF

Specifying LAXIF allows IF, WHILE, UNTIL, and WHEN clauses to evaluate to other than BIT(1) NONVARYING. NOLAXIF allows IF, WHILE, UNTIL, and WHEN clauses to evaluate to only BIT(1) NONVARYING.

The following are illegal under NOLAXIF:

```
dcl i fixed bin;
dcl b bit(8);
...
if i then ...
if b then ...
```

LAXLINK or NOLAXLINK

LAXLINK causes the compiler to ignore entry assignments where the source and target have either different linkages or different specifications of the options DESCRIPTOR, NODESCRIPTOR, ASM, COBOL or FORTRAN.

NOLAXLINK is recommended. NOLAXLINK is not the default to decrease the number of new error message that appear when you compile 370 code.

LAXMARGINS or NOLAXMARGINS

Specifying NOLAXMARGINS causes the compiler to flag any line containing non-blank characters after the right margin. This can be useful in detecing code, such as a closing comment, that has accidentally been pushed out into the right margin.

LAXPUNC | 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 or NOLAXQUAL
Specifying NOLAXQUAL causes the compiler to flag any reference to structure members that are not level 1 and are not dot qualified. Consider the following example:

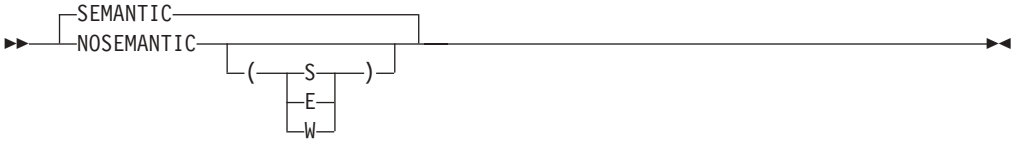
```
dcl
  1 a,
    2 b fixed bin,
    2 c fixed bin;

c   = 15;   /* would be flagged */
a.c = 15;   /* would not be flagged */
```

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.

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.

SOURCE

The SOURCE option specifies that a listing of the source input to the compiler be created.

ABBREVIATIONS: S, NS

SOURCE
The complier 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.

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.

SPILL

This option specifies register allocation spill area.

size
The value, in bytes, for the register allocation spill area.

If your program is very complex or there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

DEFAULT: SPILL(512)

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.

ABBREVIATIONS: SYN, NSYN

SYNTAX
Equivalent to NOSYNTAX(S).

NOSYNTAX
No syntax checking is performed.

NOSYNTAX(S)
No syntax checking is performed if a severe error or unrecoverable error has been detected.

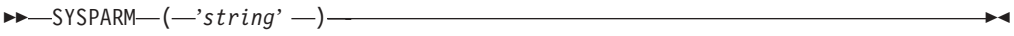
NOSYNTAX(E)
No syntax checking is performed if an error, severe error, or unrecoverable error has been detected.

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.



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 *strings* in step 2 on page 43 under "Rules for using compile-time options".

For more information about the macro facility, see the *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.

AIX
Specifies that the program runs under AIX.

CICS
Specifies that the program runs under CICS.

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

TERMINAL

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

ABBREVIATIONS: TERM, NTERM

TERMINAL
Messages are displayed on the terminal.

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

TEST

The TEST option specifies the level of testing capability 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.

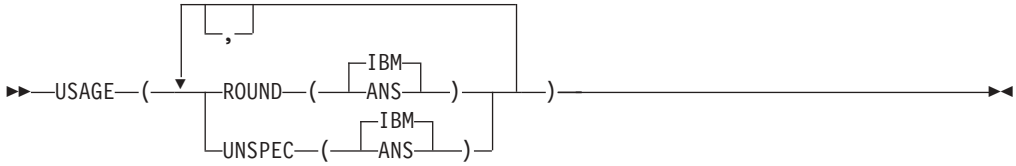
The TEST option implies GONUMBER. Because the TEST option can increase the size of the object code and can affect performance, you might want to limit the number and placement of hooks.

NOTEST
Suppresses the generation of all testing information.

TEST
Specifies that testing information should be included in the object code.

USAGE

The USAGE option lets you choose IBM or ANS semantics for selected built-in functions.



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 *OS PL/I Version 2 Language Reference*.

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.

WIDECHAR

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(BIGENDIAN)

WINDOW

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

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, 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

Under batch, this file is written to the file specified by the SYSDEFSD DD statement. Under Unix Systems Services, this file is written to the same directory as the object deck and has the extension ″def″.

For instance, given the program:

```
defs: proc;
   dcl (b,c) ext entry;
   dcl x ext fixed bin(31) init(1729);
   dcl y ext fixed bin(31) reserved;
   call b(y);
   fetch c;
   call c;
end;
```

The following def file would be produced:

```
EXPORTS CODE
  DEFS
EXPORTS DATA
  X
IMPORTS
  B
  Y
FETCH
  C
```

The def file can be used to be build a dependency graph or cross-reference analysis of your application.

NODEF
No definition side-deck file is created.

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

Under batch, this file is written to the file specified by the SYSXMLSD DD statement. Under Unix Systems Services, this file is written to the same directory as the object deck and has the extension ″xml″.

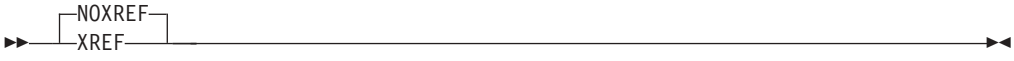
The DTD file for the XML produced is:

```
<?xml encoding="UTF-8"?>

<!ELEMENT PACKAGE ((PROCEDURE)*,(MESSAGE)*,FILEREFERNCETABLE)>
<!ELEMENT PROCEDURE (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT BEGINBLOCK (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT MESSAGE (MSGNUMBER,MSGLINE?,MSGFILE?,MSGTEXT)>
<!ELEMENT FILE (FILENUMBER,INCLUDEDFROMFILE?,INCLUDEDONLINE?,FILENAME)>
<!ELEMENT FILEREFERENCETABLE (FILECOUNT,FILE+)>

<!ELEMENT BLOCKFILE (#PCDATA)>
<!ELEMENT BLOCKLINE (#PCDATA)>
<!ELEMENT MSGNUMBER (#PCDATA)>
<!ELEMENT MSGLINE (#PCDATA)>
<!ELEMENT MSGFILE (#PCDATA)>
<!ELEMENT MSGTEXT (#PCDATA)>
<!ELEMENT FILECOUNT (#PCDATA)>
<!ELEMENT FILENUMBER (#PCDATA)>
<!ELEMENT FILENAME (#PCDATA)>
<!ELEMENT INCLUDEFROMFILE (#PCDATA)>
<!ELEMENT INCLUDEDONLINE (#PCDATA)>
```

NOXML
No XML side-file is created.

XREF

The XREF option provides a cross-reference table of names used in the program together with the numbers of the statements in which they are declared or referenced in the compiler listing.



ABBREVIATIONS: X, NX

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 111. If both XREF and ATTRIBUTES are specified, the two listings are combined.

Chapter 7. PL/I preprocessors

Include preprocessor 84
Examples: 84
Include preprocessor options in the configuration file 84
Macro preprocessor. 85
Macro preprocessor options 85
Macro facility options in the configuration file. . 86
SQL support 87
Programming and compilation considerations . . 87
SQL preprocessor options. 87
Abbreviations: 89
SQL preprocessor options in the configuration file 92
Coding SQL statements in PL/I applications . . 93
Defining the SQL communications area . . . 93
Defining SQL descriptor areas 93
Embedding SQL statements 94
Using host variables 95
Determining equivalent SQL and PL/I data types 96
Large Object (LOB) support 98
General information on LOBs 98
PL/I variable declarations for LOB Support . 99
Sample programs for LOB support 100
User defined functions sample programs . . . 100
Determining compatibility of SQL and PL/I data types 102
Using host structures. 102
Using indicator variables 103
Host structure example 104
CONNECT TO statement 104
DECLARE TABLE statement 105
DECLARE STATEMENT statement 105
Logical NOT sign (¬) 105
Handling SQL error return codes. 105
Using the DEFAULT(EBCDIC) compile-time option. 106
SQL compatibility and migration considerations 106
CICS support 107
Programming and compilation considerations 107
CICS preprocessor options 108
Abbreviations: 109
Coding CICS statements in PL/I applications 109
Embedding CICS statements 109
Writing CICS transactions in PL/I 110

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.

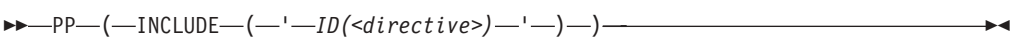
- The include preprocessor processes special include directives and incorporates external source files.
- The macro facility, based on %statements and macros, modifies your source program.
- The SQL preprocessor modifies your source program and translates EXEC SQL statements into PL/I statements.
- The 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:



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 options in the configuration file

You can set the default options for the include preprocessor by using the **pliopt** or **incopt** attribute in the configuration file. See “Configuration file attributes” on page 32.

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:

FIXED
This option specifies the default base for FIXED variables.

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

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

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

ASIS
the input text is left ″as is″.

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

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

UPPER
rescans will not be case-sensitive.

ASIS
rescans will be case-sensitive.

To see the effect of this option, consider the following code fragment

```
%dcl eins char ext;
%dcl text char ext;

%eins = 'zwei';

%text = 'EINS';
display( text );

%text = 'eins';
display( text );
```

When 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

```
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 );
```

In short: RESCAN(UPPER) ignores case while RESCAN(ASIS) does not.

Macro facility options in the configuration file

You can set the default options for the macro facility by using the **pliopt** or **macopt** attribute in the configuration file. See "Configuration file attributes" on page 32.

SQL support

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 AIX.

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 PL/I SQL support. The following table describes these considerations.

Table 6. Considerations for EXEC SQL support

If the target system is...	Use this compile-time option...
Using DB2 in native AIX mode	DEFAULT (ASCII IEEE)
Using DB2 in z/OS emulation mode	DEFAULT (EBCDIC HEXADEC)

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:

```
pli sqlprog.pli -ldb2
```

The name of the DB2 library is*libdb2.a*. In the sample above, the first three characters of the library name (*lib*) and the library name extension (*.a*) are assumed.

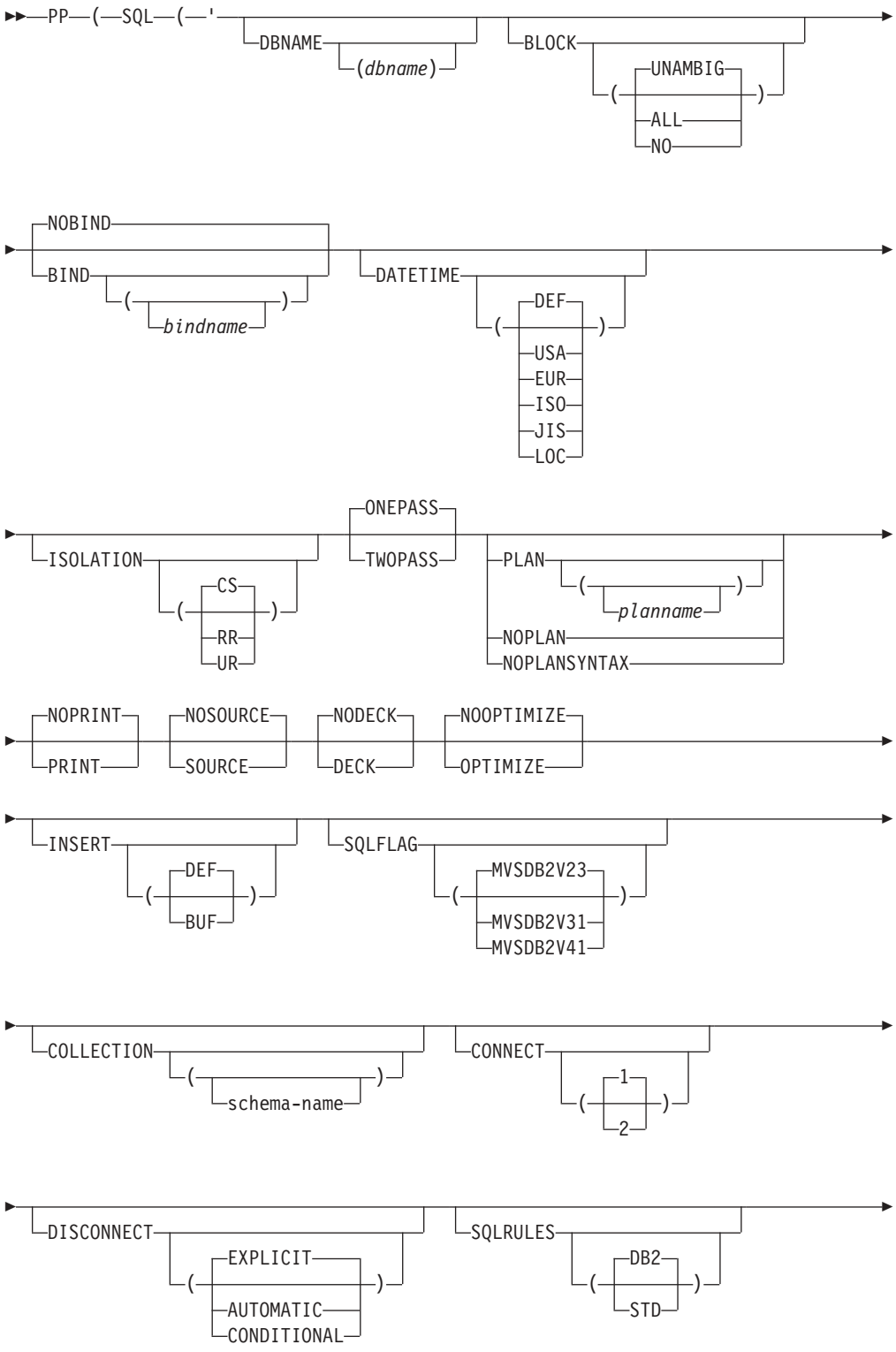
SQL Users

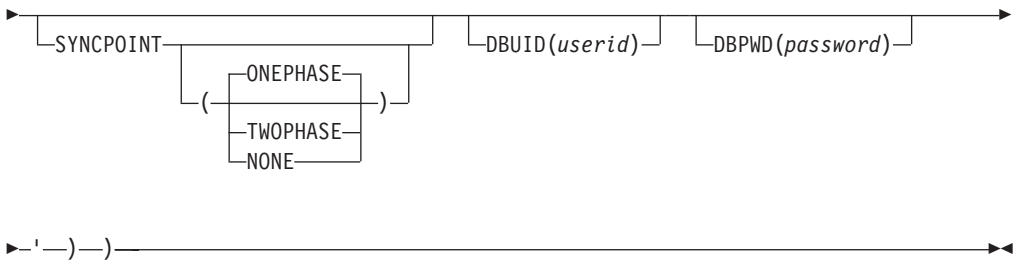
You must have DB2 for AIX installed and started before you can compile a program containing EXEC SQL statements. To find out how to install DB2, refer to database installation guide for the platform you are using.

SQL preprocessor options

The following syntax diagram illustrates all of the options supported by the SQL preprocessor.

SQL support


PP
SQL
'
DBNAME
(dbname)
BLOCK
UNAMBIG
ALL
NO
NOBIND
BIND
(bindname)
DATETIME
DEF
USA
EUR
ISO
JIS
LOC
ONEPASS
TWOPASS
ISOLATION
CS
RR
UR
PLAN
(planname)
NOPLAN
NOPLANSYNTAX
NOPRINT
PRINT
NOSOURCE
SOURCE
NODECK
DECK
NOOPTIMIZE
OPTIMIZE
INSERT
DEF
BUF
SQLFLAG
MVSDB2V23
MVSDB2V31
MVSDB2V41
COLLECTION
schema-name
CONNECT
1
2
DISCONNECT
EXPLICIT
AUTOMATIC
CONDITIONAL
SQLRULES
DB2
STD



Abbreviations:

DB, BLK, DT, ISOL, ON, TW, S, NS, D, ND, OPT, NOPT, 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.

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

ISOLATION
Specifies the isolation level at which your program runs.

CS Cursor stability

RR Repeatable read

UR Uncommitted read

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.

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.

PRINT or NOPRINT
Specifies whether or not the source code generated by the SQL preprocessor is printed in the source listing(s) produced by subsequent preprocessors or the compiler.

SOURCE or NOSOURCE
Specifies whether or not the source input to the SQL preprocessor is printed.

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.

OPTIMIZE or NOOPTIMIZE
If you specify OPTIMIZE, SQLDA initialization is optimized for SQL statements that use host variables. Do not specify this option when using AUTOMATIC host variables or in other situations when the address of the host variable might change during the execution of the program. (NOOPTIMIZE) is the default.

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.

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.

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.

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().

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().

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().

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.

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.

SQL preprocessor options in the configuration file

You can set the default options for the SQL preprocessor by using the **pliopt** or **sqlopt** attribute in the configuration file. See “Configuration file attributes” on page 32.

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* (SBOF-8923). 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 2 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 sqlcaid     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 sqlerrd(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 sqlext,
      3 sqlwarn8  char(1),
      3 sqlwarn9  char(1),
      3 sqlwarna  char(1),
      3 sqlstate  char(5);       /* State corresponding to SQLCODE */
```

Figure 2. 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.

```
Dcl
  1 Sqlda based(Sqldaptr),
    2 sqldaid     char(8),          /* Eye catcher = 'SQLDA   ' */
    2 sqldabc     fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
    2 sqln        fixed binary(15), /* Number of SQLVAR elements*/
    2 sqld        fixed binary(15), /* # of used SQLVAR elements*/
    2 sqlvar(Sqlsize refer(sqln)),  /* Variable Description     */
      3 sqltype   fixed binary(15), /* Variable data type       */
      3 sqllen    fixed binary(15), /* Variable data length     */
      3 sqldata   pointer,          /* Pointer to variable data value*/
      3 sqlind    pointer,          /* Pointer to Null indicator*/
      3 sqlname   char(30) var ;    /* Variable Name            */
dcl Sqlsize   fixed binary(15);     /* number of sqlvars (sqln) */
dcl Sqldaptr  pointer;
```

Figure 3. The PL/I declaration of an SQL descriptor area

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.

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. If ONEPASS is in effect, a host variable used in an SQL statement must be declared prior to the first use of the host variable in an SQL statement. In addition:

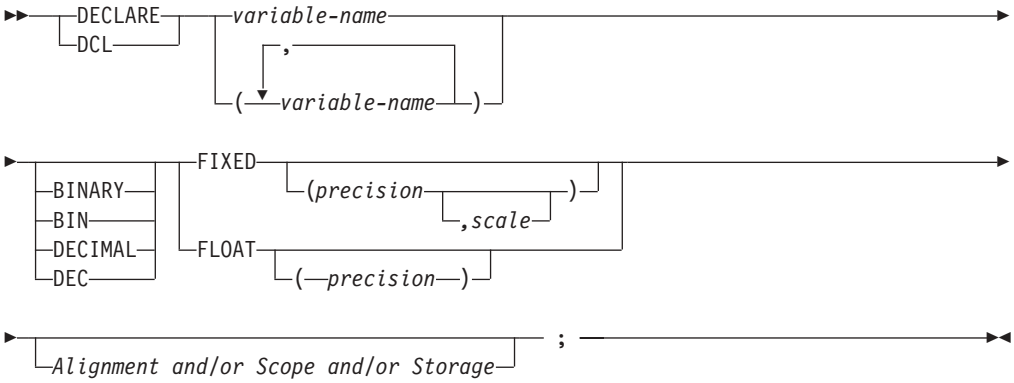
- All host variables within an SQL statement must be preceded by a colon (:).
- The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.
- An SQL statement that uses a host variable must be within the scope of the statement in which the variable was declared.
- Host variables cannot be declared as an array, although an array of indicator variables is allowed when the array is associated with a host structure.

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. 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”.

Only the names and data attributes of the variables are used by the preprocessor; the alignment, scope, and storage attributes are ignored.

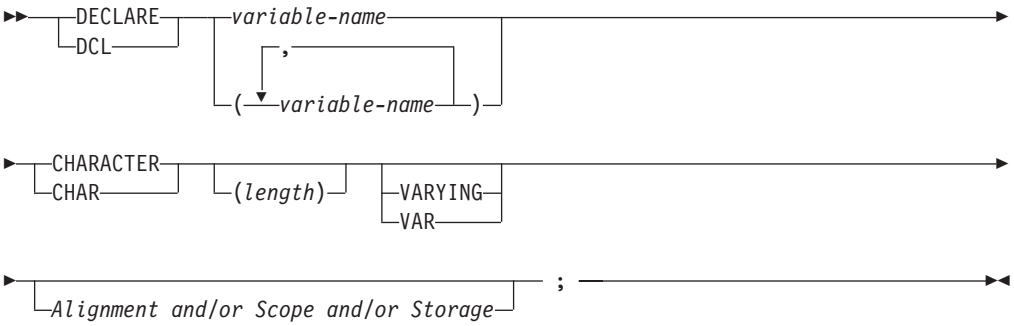
Numeric host variables: The following figure shows the syntax for valid numeric host variable declarations.



Notes

- BINARY/DECIMAL and FIXED/FLOAT can be specified in either order.
- The precision and scale attributes can follow BINARY/DECIMAL.
- A value for *scale* can only be specified for DECIMAL FIXED.

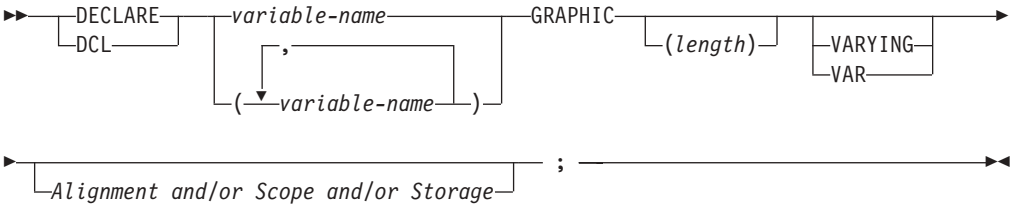
Character host variables: The following figure shows the syntax for valid character host variables.



Notes

- For non-varying character host variables, *length* must be a constant no greater than the maximum length of SQL CHAR data.
- For varying-length character host variables, *length* must be a constant no greater than the maximum length of SQL LONG VARCHAR data.

Graphic host variables: The following figure shows the syntax for valid graphic host variables.



Notes

- For non-varying graphic host variables, *length* must be a constant no greater than the maximum length of SQL GRAPHIC data.
- For varying-length graphic host variables, *length* must be a constant no greater than the maximum length of SQL LONG VARGRAPHIC data.

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 7. SQL data types generated from PL/I declarations

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
BIN FIXED(n), n < 16	500	2	SMALLINT
BIN FIXED(n), n ranges from 16 to 31	496	4	INTEGER
DEC FIXED(p,s)	484	p (byte 1) s (byte 2)	DECIMAL(p,s)
BIN FLOAT(p), 22 ≤ p ≤ 53	480	8	FLOAT
DEC FLOAT(m), 7 ≤ m ≤ 16	480	8	FLOAT
CHAR(n), 1 ≤ n ≤ 254	452	n	CHAR(n)

Table 7. SQL data types generated from PL/I declarations (continued)

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
CHAR(n) VARYING, 1 ≤ n ≤ 4000	448	n	VARCHAR(n)
CHAR(n) VARYING, n > 4000	456	n	LONG VARCHAR
GRAPHIC(n), 1 ≤ n ≤ 127	468	n	GRAPHIC(n)
GRAPHIC(n) VARYING, 1 ≤ n ≤ 2000	464	n	VARGRAPHIC(n)
GRAPHIC(n) VARYING, n > 2000	472	n	LONG VARGRAPHIC

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 8. SQL data types mapped to PL/I declarations

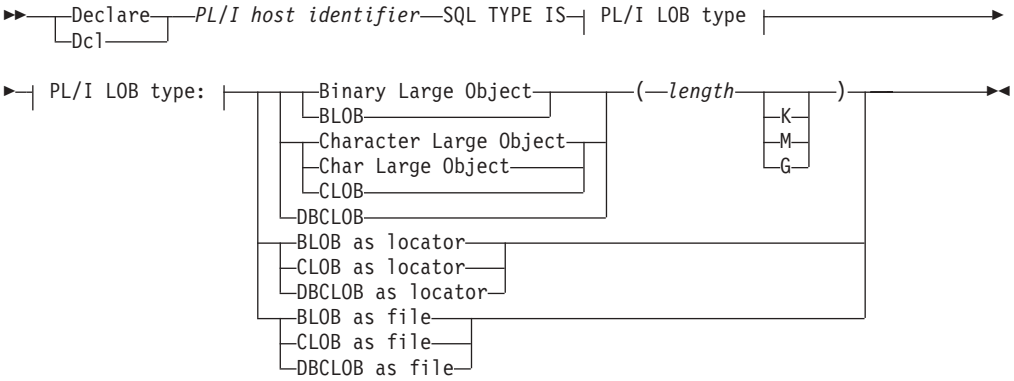
SQL Data Type	PL/I Equivalent	Notes
SMALLINT	BIN FIXED(15)	
INTEGER	BIN FIXED(31)	
DECIMAL(p,s)	DEC FIXED(p) or DEC FIXED(p,s)	p = precision and s = scale; 1 ≤ p ≤ 31 and 0 ≤ s ≤ p
FLOAT	BIN FLOAT(p) or DEC FLOAT(m)	22 ≤ p ≤ 53 7 ≤ m ≤ 16
CHAR(n)	CHAR(n)	1 ≤ n ≤ 254
VARCHAR(n)	CHAR(n) VAR	1 ≤ n ≤ 4000
LONG VARCHAR	CHAR(n) VAR	n > 4000
GRAPHIC(n)	GRAPHIC(n)	*n* is a positive integer from 1 to 127 that refers to the number of double-byte characters, not to the number of bytes
VARGRAPHIC(n)	GRAPHIC(n) VAR	*n* is a positive integer that refers to the number of double-byte characters, not to the number of bytes; 1 ≤ n ≤ 2000
LONG VARGRAPHIC	GRAPHIC(n) VAR	n > 2000
DATE	CHAR(n)	*n* must be at least 10
TIME	CHAR(n)	*n* must be at least 8
TIMESTAMP	CHAR(n)	*n* must be at least 26

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 now recognized by the preprocessor. Refer to the DB2 manuals for more information on these subjects,

General information on LOBs

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). BLOBS, CLOBS, AND DBCLOBS can be declared in PL/I programs with the following syntax (*PL/I variables for Large Object columns, locators, and files*):



BLOB, CLOB, and DBCLOB data types

The variable declarations for BLOBs, CLOBs, and DBCLOBs are transformed by the PL/I SQL preprocessor.

For example, consider the following declare:

```
Dcl my-identifier-name SQL TYPE IS lob-type-name (length);
```

The SQL preprocessor would transform the declare into this structure:

```
Define structure
  1  lob-type-name_length,
    2  Length unsigned fixed bin(31),
    2  Data(length) char(1);
Dcl my-identifier-name TYPE lob-type-name_length;
```

In this structure, `my-identifier-name` is the name of your PL/I host identifier and `lob-type-name_length` is a name generated by the preprocessor consisting of the LOB type and the length.

For DBCLOB data types, the generated structure looks a little different:

```
Define structure
  1  lob-type-name_length,
    2  Length unsigned fixed bin(31),
    2  Data(length) type wchar_t;
```

In this case, type `wchar_t` is defined in the include member `sqlsystm.cpy`. This member must be included to use the DBCLOB data type.

length

The length field is an unsigned integer that maps to a fixed binary. The values of the length field can range from 0 to (2**32)-8. If the length field is appended by a K, M, or G, then the length is calculated by the preprocessor.

BLOB, CLOB, and DBCLOB LOCATOR data types

The variable declarations for BLOB, CLOB, and DBCLOB locators are also transformed by the PL/I SQL preprocessor.

For example, consider the following declare:

```
Dcl my-identifier-name SQL TYPE IS lob-type AS LOCATOR;
```

The SQL preprocessor would transform this declare into the following code:

```
Define alias lob-type_LOCATOR fixed bin(31) unsigned;

Dcl my-identifier-name TYPE lob-type_LOCATOR;
```

In this case, `my-identifier-name` is your PL/I host identifier and `lob-type_LOCATOR` is a name generated by the preprocessor consisting of the LOB type and the string LOCATOR.

BLOB, CLOB, and DBCLOB FILE data types

The variable declarations for BLOB, CLOB, and DBCLOB files are also transformed by the PL/I SQL preprocessor.

For example, consider this declare:

```
Dcl my-identifier-name SQL TYPE IS lob-type AS FILE;
```

The SQL preprocessor transforms the declare as follows:

```
Define structure
   1  lob-type_FILE,
     2  Name_Length unsigned fixed bin(31),
     2  Data_Length unsigned fixed bin(31),
     2  File_Options unsigned fixed bin(31),
     2  Name char(255);

Dcl my-identifier-name TYPE lob-type_FILE;
```

Again, `my-identifier-name` is your PL/I host identifier and `lob-type_FILE` is a name generated by the preprocessor consisting of the LOB type and the string FILE.

PL/I variable declarations for LOB Support

The following examples provide sample PL/I variable declarations and their corresponding transformations for LOB support.

Example 1:

```
Dcl my_blob SQL TYPE IS blob(2000);
```

After transform:

```
Define structure
   1  blob_2000,
     2  Length unsigned fixed bin(31),
     2  Data(2000) char(1);
Dcl my_blob type blob_2000;
```

Example 2:

```
Dcl my_dbclob SQL TYPE IS DBCLOB(1M);
```

After transform:

```
Define structure
   1  dbclob_1m,
     2  Length unsigned fixed bin(31),
     2  Data(1048576) type wchar_t;
Dcl my_dbclob type dbclob_1m ;
```

Example 3:

```
Dcl my_clob_locator SQL TYPE IS clob as locator;
```

After transform:

```
 Define alias clob_locator fixed bin(31) unsigned;
 Dcl my_clob_locator type clob_locator;
```

Example 4:

```
Dcl my_blob_file SQL TYPE IS blob as file;
```

After transform:

```
Define structure
   1  blob_FILE,
     2  Name_Length unsigned fixed bin(31),
     2  Data_Length unsigned fixed bin(31),
     2  File_Options unsigned fixed bin(31),
     2  Name char(255);

Dcl my_blob_file type blob_file;
```

Example 5:

```
Dcl my_dbclob_file SQL TYPE IS dbclob as file;
```

After transform:

```
Define structure
   1  dbclob_FILE,
     2  Name_Length unsigned fixed bin(31),
     2  Data_Length unsigned fixed bin(31),
     2  File_Options unsigned fixed bin(31),
     2  Name char(255);

Dcl my_dbclob_file type dbclob_file;
```

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.

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. If you are using PL/I for AIX, for example, you would copy udfdll to /u/inst1/sqllib/function if that were the user defined function directory on your AIX machine 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 bldaddudf 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
Fetches `ID` and `Dept` from the STAFF table then adds them together by calling `MyAdd` UDF. Use the command file `bldmyadd` to compile and link it.

UDFMYDIV.PLI
Fetches `ID` and `Dept` from the STAFF table then divides them by calling `MyDiv` UDF. Use the command file `bldmydiv` to compile and link it.

UDFMYUP.PLI
Fetches `Name` from the STAFF table then calls `MyUpper` to change the vowels to uppercase. Use the command file `bldmyup` to compile and link it.

UDFMYCNT.PLI
Fetches `ID` from the STAFF table, outputs the count of the call, then divides `ID` by the count. Use the command file `bldmycnt` to compile and link it.

UDFCLOB.PLI
Fetches the resume for employee '000150' then calls `ClobUpper` to change the vowels to uppercase. Use the command file `bldclobu` 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.

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:

```
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.

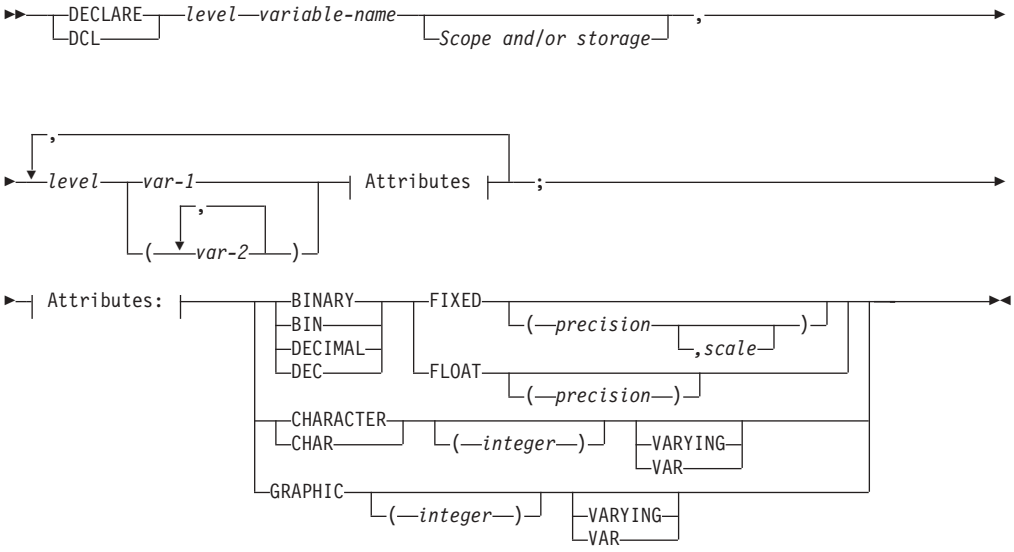
Host structures are limited to two levels. A host structure can be thought of as a named collection of host variables.

You must terminate the host structure variable by ending the declaration with a semicolon. For example:

```
dcl 1 A,
    2 B char,
    2 (C, D) char;
dcl (E, F) char;
```

Host variable attributes can be specified in any order acceptable to PL/I. For example, BIN FIXED(31), BINARY FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

The following diagram shows the syntax for valid host structures.



Using indicator variables

An indicator variable is a two-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.

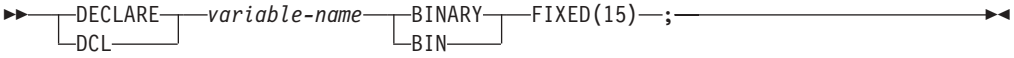
Given the statement:

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

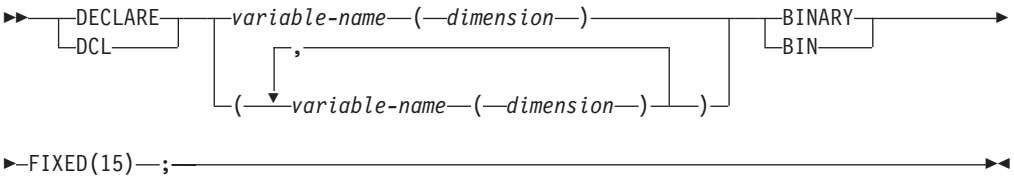
Variables can be declared as follows:

```
exec sql begin declare section;
dcl Cls_Cd    char(7);
dcl Day       bin fixed(15);
dcl Bgn       char(8);
dcl End       char(8);
dcl (Day_Ind, Bgn_Ind, End_Ind)   bin fixed(15);
exec sql end declare section;
```

The following diagram shows the syntax for a valid indicator variable.



The following diagram shows the syntax for a valid indicator array.



Host structure example

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

```
dcl 1 games,
      5 sunday,
        10 opponents char(30),
        10 gtime     char(10),
        10 tv        char(6),
        10 comments  char(120) var;
dcl indicator(4) fixed bin (15);


exec sql
     fetch cursor_a
     into :games.sunday.opponents:indicator(1),
          :games.sunday.gtime:indicator(2),
          :games.sunday.tv:indicator(3),
          :games.sunday.comments:indicator(4);


exec sql
     fetch cursor_a
     into :games.sunday:indicator;
```

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.

Logical NOT sign (¬)

The preprocessor performs the following translations within SQL statements:

¬= is translated to <>

¬< is translated to >=

¬> is translated to <=

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.

- If you are using DSNTIAR in native AIX PL/I programs, DSNTIAR must be compiled with the DEFAULT(ASCII) and SYSTEM(AIX) compile-time options.
- If you are using DSNTIAR in host emulation PL/I programs, DSNTIAR must be compiled with the DEFAULT(EBCDIC) compile-time option.

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:

```
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.

Using the DEFAULT(EBCDIC) compile-time option

When you use the compile-time option DEFAULT(EBCDIC) with SQL statements that contain input or output character host variables, the SQL preprocessor inserts extra code in the expansion for the SQL statements to convert character data between ASCII and EBCDIC unless the character data has the FOR BIT DATA column attribute.

Avoiding automatic conversion for specific character data: If you do not want data to be converted, you have to give explicit instructions to the preprocessor. For example, if you did not want conversion to occur between a CHARACTER variable and a FOR BIT DATA column, you could include a PL/I comment as shown in the following example:

```
dcl SL1 /* %ATTR FOR BIT DATA */ char(9);
```

The first nonblank character in the comment must be a percent (%) sign followed by the keywords ATTR FOR BIT DATA.

You can put this comment anywhere after the variable name as long as it appears before the end of the declaration for that variable. Neither SL2 nor SL4 are converted in the following example:

```
Dcl SL2 /* %ATTR FOR BIT DATA */ char(9),
    SL3 char (20); /* %ATTR FOR BIT DATA */
Dcl (SL4 /* %ATTR FOR BIT DATA */,
    SL5) char (9);
```

Using the DEFAULT(NONNATIVE) compile-time option: When you use the compile-time option DEFAULT(NONNATIVE) with an SQLDA that describes a decimal field, you must re-reverse the SQLLEN field after the conversion done by the SQL preprocessor.

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 AIX for eventual execution under CICS on AIX or under CICS/ESA, CICS/MVS, or CICS/VSE systems on S/390.

Before you can compile programs containing EXEC CICS statements, you must have TXseries for Multiplatforms Version 5 or later installed. 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.
- You must use the PP(CICS(*options*) MACRO) 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(...) )
```

TXseries for Multiplatforms supplies a command `cicstcl` which compiles and links a CICS PL/I program with all of the correct CICS options. The CICS include directory is automatically specified. The command line syntax for `cicstcl` is as follows:

```
cicstcl -lIBMPLI program.pli
```

If you choose not to use `cicstcl` to compile your CICS PL/I programs, then you must specify `-I/usr/lpp/cics/include` on the `pli` command line so that the compiler can find the CICS include files.

If you want to compile a CICS and DB2 PL/I program named *cicsdb2.pli*, you would use the following command:

```
pli -I/usr/lpp/cics/include
-qsystem=CICS
-qpp=CICS=noedf:nodebug:nosource:noprint:MACRO
-o cicsdb2.ibmpli
-bl:/usr/lpp/cics/lib/cicsprIBMPLI.exp
-eplicics
-L/usr/lib/dce
-ldcelibc_r
-ldcepthreads
-ldb2
-lplishr_r
-lc_r
cicsdb2.pli
```

All the PL/I stream output produced in one of the following ways is written to the CPLI transient data queue (TDQ) and it will be routed to the system console file:

- PUT statements to SYSPRINT
- Messages written to the MSGFILE
- DISPLAY statements

Output produced by PLIDUMP is always written to the CPLD transient data queue and it is routed to the system console file which can be found in /var/CICS_regions/<*RegionName*>/console.msg, where <*RegionName*> is user defined.

The full TXseries for Multiplatforms API is supported for PL/I programs, see *TXseries for Multiplatforms, CICS Application Programming Reference*, SC09-4461. Support is also provided for PL/I programs to use External Call Interface (ECI).

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.

You also need to consider options depending on the nature of your program and which CICS system is used for executing the program.

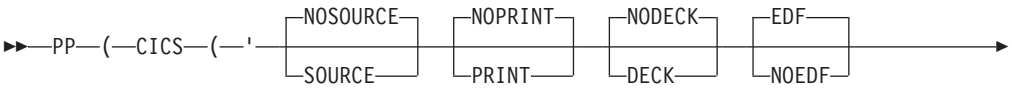
Table 9. Considerations for EXEC CICS support

If you are using ...	Use compile-time option(s)...
CICS for Windows	PP(CICS MACRO)
CICS Files containing native data	DEFAULT (ASCII NATIVE IEEE) as appropriate
UDB in native mode	DEFAULT (ASCII NATIVE IEEE) as appropriate

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.



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).

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*, SC09-4460. 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.

- 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
- You should 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.

  COBOL and C are supported under CICS by the following IBM PL/I products:
  IBM Enterprise PL/I for z/OS
  IBM PL/I for AIX
  IBM WebSphere Studio PL/I for Windows
  IBM PL/I MVS and VM
- Do not use the PLISRTx built-in subroutines.
- Do not make calls to IMS using the PLITDLI, ASMTDLI, or EXEC DLI.

Chapter 8. Compilation output

Compilation output

Using the compiler listing . . . . . 111 Compiler output files. 121

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 121 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 to your terminal. The following description of the listing refers to its appearance on a printed page.

This listing for the TOWERS program highlights some of the more useful parts of the compiler listing.

```
33H1858  IBM(R) PL/I for AIX      2.0.0.0  (Built:20040512)         2004.05.18 17:53:18   Page     1

                   Options Specified 1


 Environment: gn xref(full) source
 Command: options nest aggregate source
 Line.File Process Statements

    1.1    *PROCESS LANGLVL(SAA2) ATTRIBUTES(FULL);
    2.1    *PROCESS NOT('^') OR('|') GN LINECOUNT(55);
    3.1    *PROCESS PP( MACRO('rescan(upper)') ) NEST XREF(FULL);
    4.1    *PROCESS LIMITS(EXTNAME(31));
```

Figure 4. TOWERS program compiler listing (Part 1 of 8)

```
33H1858  IBM(R) PL/I for AIX                                    2004.05.18 17:53:18   Page     2
                   Options Used 2

 +    AGGREGATE(DECIMAL)
 +    ATTRIBUTES(FULL)
      BIFPREC(31)
      BLANK('09'x)
      CHECK( NOSTORAGE )
      CMPAT(LE)
      CODEPAGE(00819)
    NOCOMPILE(S)
    NOCOPYRIGHT
      CURRENCY('$')
      DEFAULT(IBM ASSIGNABLE NOINITFILL NONCONNECTED LOWERINC
              DESCRIPTOR DESCLIST DUMMY(ALIGNED) ORDINAL(MIN)
              BYADDR RETURNS(BYVALUE) NORETCODE
              NOINLINE ORDER NOOVERLAP NONRECURSIVE ALIGNED
              NULLSYS EVENDEC SHORT(HEXADEC)
              ASCII IEEE NATIVE NATIVEADDR E(IEEE))
    NOEXIT
      EXTRN(SHORT)
      FLAG(W)
      FLOATINMATH(ASIS)
 +    GONUMBER
    NOGRAPHIC
      IMPRECISE
      INCAFTER(PROCESS(""))
    NOINCDIR
    NOINITAUTO
    NOINITBASED
    NOINITCTL
    NOINITSTATIC
    NOINSOURCE
    NOINTERRUPT
      LANGLVL(SAA2 NOEXT)
 +    LIMITS( EXTNAME(31) FIXEDBIN(31,31) FIXEDDEC(15,15) NAME(100) )
 +    LINECOUNT(55)
    NOLIST
    NOMACRO
      MARGINI(' ')
      MARGINS(2,72)
      MAXMEM(1048576)
      MAXMSG(W 250)
      MAXSTMT(4096)
      MAXTEMP(50000)
    NOMDECK
      MSG(*)
      NAMES('@#$' '@#$')
      NATLANG(ENU)
 +    NEST
 +    NOT('^')
      NUMBER
      OBJECT
    NOOFFSET
```

Figure 4. TOWERS program compiler listing (Part 2 of 8)

```
33H1858  IBM(R) PL/I for AIX                                    2004.05.18 17:53:18   Page     3

      OPTIMIZE(0)
 +    OPTIONS
      OR('|')
 +    PP( MACRO )
    NOPPTRACE
      PRECTYPE(ANS)
      PREFIX(CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW
             NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE
             UNDERFLOW ZERODIVIDE)
    NOPROCEED(S)
      PROCESS
      REDUCE
      RESEXP
      RESPECT()
      RULES(IBM BYNAME EVENDEC GOTO NOLAXBIF NOLAXCTL LAXDCL NOLAXDEF LAXIF
            LAXLINK LAXMARGINS LAXPUNC LAXQUAL NOLAXSTRZ MULTICLOSE)
    NOSEMANTIC(S)
 +    SOURCE
      SPILL(512)
      STATIC(SHORT)
    NOSTMT
    NOSTORAGE
    NOSYNTAX(S)
      SYSPARM('')
      SYSTEM(AIX)
      TERMINAL
    NOTEST
      USAGE( ROUND(IBM) UNSPEC(IBM) )
      WIDECHAR(BIGENDIAN)
      WINDOW(1950)
      XINFO(NODEF NOXML)
 +    XREF(FULL)
```

Figure 4. TOWERS program compiler listing (Part 3 of 8)

```
33H1858  IBM(R) PL/I for AIX                                  2004.05.18 17:53:18   Page     4
 Compiler Source
 Line.File LV NT
    5.1
    6.1           /*********************************************************************/
    7.1           /*                                                                   */
    8.1           /*  NAME - TOWERS.PLI                                                */
    9.1           /*                                                                   */
   10.1           /*  DESCRIPTION                                                      */
   11.1           /*    Towers of Hanoi program.                                       */
   12.1           /*                                                                   */
   13.1           /*    Licensed Materials - Property of IBM                           */
   14.1           /*    5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,1996.          */
   15.1           /*    All Rights Reserved.                                           */
   16.1           /*    US Government Users Restricted Rights-- Use, duplication or    */
   17.1           /*    disclosure restricted by GSA ADP Schedule Contract with        */
   18.1           /*    IBM Corp.                                                      */
   19.1           /*                                                                   */
   20.1           /*  DISCLAIMER OF WARRANTIES                                         */
   21.1           /*    The following ïenclosedù code is sample code created by IBM    */
   22.1           /*    Corporation. This sample code is not part of any standard      */
   23.1           /*    IBM product and is provided to you solely for the purpose of   */
   24.1           /*    assisting you in the development of your applications.  The    */
   25.1           /*    code is provided "AS IS", without warranty of any kind.        */
   26.1           /*    IBM shall not be liable for any damages arising out of your    */
   27.1           /*    use of the sample code, even if IBM has been advised of the    */
   28.1           /*    possibility of such damages.                                   */
   29.1           /*                                                                   */
   30.1           /*********************************************************************/
   31.1           TOWERS:PROC OPTIONS(MAIN);
   32.1
   33.1             /***************************************/
   34.1             /* Program variables                   */
   35.1             /***************************************/
   36.1     1       DCL TOWER(3, N )  CTL CHAR(1);
   37.1
   38.1     1       DCL TOPS(3)       FIXED BIN(31);
   39.1     1       DCL N             FIXED BIN(31);
   40.1     1       DCL NX            CHAR(10) VARYING;
   41.1     1       DCL RUN_AGAIN     CHAR(1) VARYING;
   42.1     1       DCL JX            FIXED BIN(31);
   43.1     1       DCL MOVES         FIXED BIN(31);
   44.1     1       DCL ALPHA         CHAR(*) VALUE('abcdefghijklmnopqrstuvwxyz');
   45.1
   46.1
   47.1             /***************************************/
   48.1             /* On unit to handle non-numeric data. */
   49.1             /***************************************/
   50.1     1       ON CONVERSION
   51.1     1         BEGIN;
   52.1     2           DISPLAY( 'tower size must be numeric' );
   53.1     2           GOTO RECOVER;
```

Figure 4. TOWERS program compiler listing (Part 4 of 8)

```
33H1858  IBM(R) PL/I for AIX                                    2004.05.18 17:53:18   Page     5
  Line.File LV NT
   54.1     2          END;
   55.1
   56.1     1        RECOVER:
   57.1                DISPLAY( 'How big a tower do you want?') REPLY( NX );
   58.1
   59.1     1        N = NX;
   60.1     1        IF N < 2 THEN
   61.1     1          DO;
   62.1     1  1         DISPLAY( 'tower size must be at least 2' );
   63.1     1  1         GOTO RECOVER;
   64.1     1  1       END;
   65.1     1        ELSE;
   66.1     1        N = MIN(N,7);
   67.1
   68.1     1        ALLOCATE TOWER;
   69.1
   70.1     1        DO JX = 1 TO HBOUND(TOWER,2);
   71.1     1  1       TOWER(1,JX) = SUBSTR(ALPHA,JX,1);
   72.1     1  1     END;
   73.1     1        TOPS(1) = 1;
   74.1
   75.1     1        TOWER(2,*) = '';
   76.1     1        TOPS(2) = HBOUND(TOWER,2) + 1;
   77.1
   78.1     1        TOWER(3,*) = '';
   79.1     1        TOPS(3) = HBOUND(TOWER,2) + 1;
   80.1
   81.1     1        MOVES = 0;
   82.1
   83.1     1        CALL DISPLAY_TOWERS;
   84.1
   85.1     1        CALL HANOI( HBOUND(TOWER,2),
   86.1                           1,
   87.1                           2,
   88.1                           3 );
   89.1
   90.1     1        PUT SKIP LIST( '' );
   91.1     1        PUT SKIP LIST( 'number of moves = ' || TRIM(CHAR(MOVES)) );
   92.1
   93.1     1        FREE TOWER;
   94.1
   95.1     1        PUT SKIP;
   96.1     1        FREE TOWER;
   97.1
   98.1     1        DISPLAY('Do you want to specify a new tower size?')
   99.1                 REPLY( RUN_AGAIN );
  100.1
  101.1     1        IF RUN_AGAIN = 'y' | RUN_AGAIN = 'Y' THEN
  102.1     1          GOTO RECOVER;
  103.1     1        ELSE;
  104.1
```

Figure 4. TOWERS program compiler listing (Part 5 of 8)

```
33H1858  IBM(R) PL/I for AIX                                  2004.05.18 17:53:18   Page     6
Line.File LV NT

  105.1             /*****************************************************/
  106.1             /* Use recursive algorithm to determine next move. */
  107.1             /*****************************************************/
  108.1     1       HANOI: PROC( RINGS, FROM, USING, TO ) RECURSIVE;
  109.1
  110.1     2         DCL ( RINGS, FROM, USING, TO ) FIXED BIN(31);
  111.1
  112.1     2         IF RINGS > 1 THEN
  113.1     2           CALL HANOI( RINGS-1, FROM, TO, USING );
  114.1     2         ELSE;
  115.1
  116.1     2         TOPS(TO) = TOPS(TO) - 1;
  117.1     2         TOWER(TO,TOPS(TO)) = TOWER(FROM,TOPS(FROM));
  118.1     2         TOWER(FROM,TOPS(FROM)) = '';
  119.1     2         TOPS(FROM) = TOPS(FROM) + 1;
  120.1
  121.1     2         MOVES = MOVES + 1;
  122.1
  123.1     2         CALL DISPLAY_TOWERS;
  124.1
  125.1     2         IF RINGS > 1 THEN
  126.1     2           CALL HANOI( RINGS-1, USING, FROM, TO );
  127.1     2         ELSE;
  128.1
  129.1     2       END HANOI;
  130.1
  131.1
  132.1             /*****************************************************/
  133.1             /* Show the configuration of the 3 towers.         */
  134.1             /*****************************************************/
  135.1     1       DISPLAY_TOWERS: PROC;
  136.1
  137.1     2         DCL JX            FIXED BIN(31);
  138.1
  139.1     2         PUT SKIP LIST( '' );
  140.1
  141.1     2         DO JX = 1 TO HBOUND(TOWER,2);
  142.1     2  1        PUT SKIP LIST(    '   ' || TOWER(1,JX)
  143.1                                || '   ' || TOWER(2,JX)
  144.1                                || '   ' || TOWER(3,JX) );
  145.1     2  1      END;
  146.1
  147.1     2         PUT SKIP LIST(    '   ' || '_'
  148.1                               || '   ' || '_'
  149.1                               || '   ' || '_' );
  150.1
  151.1     2       END DISPLAY_TOWERS;
  152.1
  153.1     1     END TOWERS;
  154.1
```

Figure 4. TOWERS program compiler listing (Part 6 of 8)

```
33H1858  IBM(R) PL/I for AIX                          2004.05.18 17:53:18   Page     7

                Attribute/Xref Table  4


  Line.File Identifier                   Attributes
 44.1    ALPHA                          CONSTANT CHARACTER(26)
                                        Refs: 71.1
+++++++  CHAR                           BUILTIN
                                        Refs: 91.1
135.1    DISPLAY_TOWERS                 CONSTANT
                                        ENTRY()
                                        Refs: 83.1 123.1
110.1    FROM                           PARAMETER FIXED BIN(31,0)
                                        Refs: 113.1 117.1 117.1 118.1
                                              118.1 119.1 119.1 126.1
108.1    HANOI                          CONSTANT
                                        ENTRY( FIXED BIN(31,0),
                                               FIXED BIN(31,0),
                                               FIXED BIN(31,0),
                                               FIXED BIN(31,0) )
                                        Refs: 85.1 113.1 126.1
+++++++  HBOUND                         BUILTIN
                                        Refs: 70.1 76.1 79.1 85.1 141.1
 42.1    JX                             AUTOMATIC FIXED BIN(31,0)
                                        Refs: 71.1 71.1
                                        Sets: 70.1
137.1    JX                             AUTOMATIC FIXED BIN(31,0)
                                        Refs: 142.1 142.1 142.1
                                        Sets: 141.1
+++++++  MIN                            BUILTIN
                                        Refs: 66.1
 43.1    MOVES                          AUTOMATIC FIXED BIN(31,0)
                                        Refs: 91.1 121.1
                                        Sets: 81.1 121.1
 39.1    N                              AUTOMATIC FIXED BIN(31,0)
                                        Refs: 36.1 60.1 66.1
                                        Sets: 59.1 66.1
 40.1    NX                             AUTOMATIC CHARACTER(10) VARYING
                                        UNALIGNED
                                        Refs: 59.1
                                        Sets: 56.1
 56.1    RECOVER                        CONSTANT LABEL
                                        Refs: 53.1 63.1 102.1
110.1    RINGS                          PARAMETER FIXED BIN(31,0)
                                        Refs: 112.1 113.1 125.1 126.1
 41.1    RUN_AGAIN                      AUTOMATIC CHARACTER(1) VARYING
                                        UNALIGNED
                                        Refs: 101.1 101.1
                                        Sets: 98.1
+++++++  SUBSTR                         BUILTIN
                                        Refs: 71.1
+++++++  SYSPRINT                       CONSTANT EXTERNAL FILE STREAM
```

Figure 4. TOWERS program compiler listing (Part 7 of 8)

```
33H1858  IBM(R) PL/I for AIX                                    2004.05.18 17:53:18   Page     8

          Line.File Identifier                  Attributes
                                               OUTPUT PRINT
                                               Refs: 90.1 91.1 95.1 139.1 142.1
                                                     147.1
     110.1    TO                               PARAMETER FIXED BIN(31,0)
                                               Refs: 113.1 116.1 116.1 117.1
                                                     117.1 126.1
      38.1    TOPS                             AUTOMATIC DIM(1:3) FIXED
                                               BIN(31,0)
                                               Refs: 116.1 117.1 117.1 118.1
                                                     119.1
                                               Sets: 73.1 76.1 79.1 116.1 119.1
      36.1    TOWER                            CONTROLLED DIM(1:3,1:*)
                                               CHARACTER(1) UNALIGNED
                                               Refs: 68.1 70.1 76.1 79.1 85.1
                                                     93.1 96.1 117.1 141.1
                                                     142.1 142.1 142.1
                                               Sets: 71.1 75.1 78.1 117.1 118.1
      31.1    TOWERS                           CONSTANT EXTERNAL
                                               ENTRY()
     +++++++  TRIM                             BUILTIN
                                               Refs: 91.1
                                               Refs: 113.1 126.1
33H1858  IBM(R) PL/I for AIX                                    2004.05.18 17:53:18   Page     9


                  Aggregate Length Table 5

    Line.File Dims     Offset   Total Size    Base Size  Identifier
      38.1     1           0           12            4 TOPS

33H1858  IBM(R) PL/I for AIX                                    2004.05.18 17:53:18   Page    10


File Reference Table

  File    Included From  Name
          6
     1                  towers.pli
3

Component    Return Code    Messages (Total/Suppressed)    Time 7

MACRO           0               0  /  0                  0 secs
Compiler        0               2  /  2                  1 secs

End of compilation of TOWERS
```

Figure 4. TOWERS program compiler listing (Part 8 of 8)

1 Options specified
This section of the compiler listing shows any compile-time options you specified. Options shown under `Install:` are specified in your `pli.cfg` configuration file using the `pliopt` attribute. 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.

2 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.
- Options specified using the `pliopt` attribute in the configuration file.

3 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 towers.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.

4 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 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.

5 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 10 shows the headings for the aggregate length table columns and the description of each.

Table 10. Aggregate length table headings and description

Heading	Description
Line.File	The statement number and file number in which the aggregate is declared
Offset	The byte offset of each element from the beginning of the aggregate
Total Size	The total size in bytes of the aggregate
Base Size	The size in bytes of the data type
Identifier	The name of the aggregate and the element within the aggregate

6 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 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, CICS preprocessor, and run-time environment are listed and explained in *PL/I Messages and Codes (AIX and OS/2)*.

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 11 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 11. Possible compilation output

Output	File extension	How requested (compile-time option)	
Preprocessed source text	`.dek`	DECK option of appropriate preprocessor	
Object module	`.o`	OBJECT	
Object listing	`.asm`	LIST	

Note: You always receive a `.lst` file containing the program listing.

Part 3. Running and debugging your program

Chapter 9. Testing and debugging your programs

Testing your programs 125
General debugging tips 126
PL/I debugging techniques. 127
Using the Distributed Debugger tool 127
Using compile-time options for debugging . . 127
Using footprints for debugging 128
Using dumps for debugging 129
Formatted PL/I dumps—PLIDUMP 130
SNAP dumps for trace information 133
Using error and condition handling for debugging 133
Error and condition handling terminology 133
Error handling concepts 134
System facilities 134
Language facilities. 135
ON-units for qualified and unqualified conditions 136
Conditions used for testing and debugging 136
Common programming errors. 136
Logical errors in your source programs. . . . 136
Invalid use of PL/I 137
Calling uninitialized entry variables 137
Loops and other unforeseen errors 137
Tips for dealing with loops. 138
Unexpected input/output data 138
Unexpected program termination. 138
Other unexpected program results 139
Compiler or library subroutine failure 139
System failure 140
Poor performance 140

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.

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 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:
Distributed Debugger tool
Compile-time options
Footprints
Dumps
Error and condition handling

Using the Distributed Debugger tool

Distributed Debugger is a debugger that is provided with the PL/I for AIX compiler. To be able to use the Distributed Debugger to step through a run of your program, you need to compile your program with the TEST and GONUMBER options. To invoke the debugger to examine an executable file, issue the command `idebug filename` where filename is the name of your program. Information on how to use the Distributed Debugger is available through online help panels provided with the debug tool.

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 55.

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 69.

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:

```
program: proc( ax1xcb, ak2xcb );
         dcl (ax1xcb, ax2xcb ) pointer;
         dcl
           1 xcb based,
           2 xcba13 fixed bin,...
         ak1xcb->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 72.

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 81.

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.

```
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):

```
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 179.

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):

```
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:

```
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 166.

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 56.

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:

```
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:

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:

```
export DD_PLIDUMP=mydump
```

In your PLIDUMP specification, you cannot override other options such as RECSIZE. The default device association for the file is stderr:.

PLIDUMP example: When you run the program shown in Figure 5, a formatted dump is produced as shown in Figure 6 on page 132.

```
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.

```
1    * * * PLIDUMP * * *  Date = 910623  Time = 142249090                  Page 0001

2   User identifier: Testing PLIDUMP


3                                        * * * Calling trace * * *
    IBM0092I The PL/I PLIDUMP Service was called with Traceback (T) option
    At offset +00000024 in procedure with entry ISSUEDUMP
    From offset +0000010B in procedure with entry TESTDUMP
                                         * * * End of calling trace * * *

                                    * * * File Information * * *
                        Attributes of file SYSIN
4   STREAM INPUT EXTERNAL
5   ENVIRONMENT( CONSECUTIVE RECSIZE(80) LINESIZE(0) )
6   I/O Built-in functions:  COUNT(0) ENDFILE(0)
7   I/O Buffer:          000D9008  00000000 00000000 00000000 00000000   '................'
                         000D9018  00000000 00000000 00000000 00000000   '................'
                         000D9028  00000000 00000000 00000000 00000000   '................'
                         000D9038  00000000 00000000 00000000 00000000   '................'
                         000D9048  00000000 00000000 00000000 00000000   '................'
                         000D9058  0000                                  '..'

                        Attributes of file SYSPRINT
    STREAM OUTPUT PRINT EXTERNAL
    ENVIRONMENT( CONSECUTIVE RECSIZE(124) LINESIZE(120) PAGESIZE(60) )
    I/O Built-in functions:  PAGENO(1) COUNT(1) LINENO(1)
8   I/O Buffer:          000D8008  20416243 64456647 68496A4B 6C4D6E4F   ' AbCdEfGhIjKlMnO'
                         000D8018  70517253 74557657 78597A20 0D0A0000   'pQrStUvWxYz ....'
                         000D8028  00000000 00000000 00000000 00000000   '................'
                         000D8038  00000000 00000000 00000000 00000000   '................'
                         000D8048  00000000 00000000 00000000 00000000   '................'
                         000D8058  00000000 00000000 00000000 00000000   '................'
                         000D8068  00000000 00000000 00000000 00000000   '................'
                         000D8078  00000000 00000000 00000000             '............'

                                  * * * End of File Information * * *
                                        * * * End of Dump * * * * * *
```

Figure 6. Example of PLIDUMP output

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.

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 130). 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

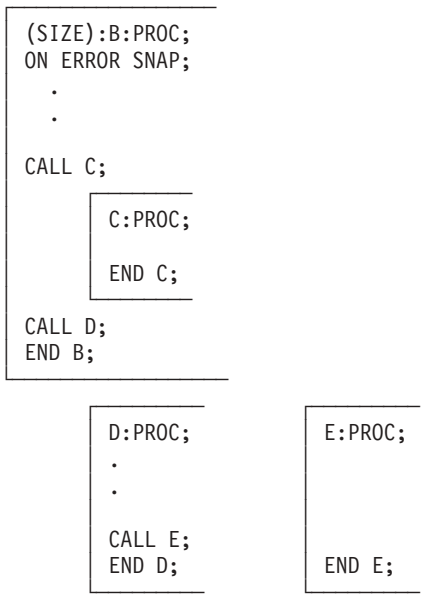
You should be familiar with several terms used in discussions of PL/I error and condition handling. The terms are listed below:

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 134 shows an example of statically and dynamically 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.

Figure 7. Static and dynamic descendant procedures

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:

```
Condition_name condition was raised
at offset xxx in procedure xxx.
```

The messages then continue as for SNAP SYSTEM messages.

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:

DATAFIELD	ONCOUNT	ONSOURCE
ONCHAR	ONFILE	ONWCHAR
ONCODE	ONGSOURCE	ONWSOURCE
ONCONDCOND	ONKEY	
ONCONDID	ONLOC	

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
- 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:

```
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*):

```
on anycondition
 begin;
  on anycondition system;
  select( oncondid() );
    when( condid_ofl )
       .
       .
       .
    when( condid_ufl )
       .
       .
       .
    when( condid_zdiv )
       .
       .
       .
```

```
    otherwise
      resignal;
  end;
end;
```

Tips for dealing with loops

To prevent a permanent loop from occurring within an ON-unit, use the following code segment:

```
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, hit **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 3, “Porting applications between platforms,” on page 9. 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:

```
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.

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 133.

- 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 data set, 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:

```
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:

```
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 136).

Other unexpected program results

Due to a difference in the way Windows responds to floating-point conditions, 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.

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 67). For additional information on improving program performance, see Chapter 15, “Improving performance,” on page 237.

Part 4. Input and output

Chapter 10. Using data sets and files

Types of data sets 143
Native data sets 144
Conventional text files and devices 144
Fixed-length data sets 144
Additional data sets 145
Varying-length data sets 145
Regional data sets 145
Workstation VSAM data sets 145
Establishing data set characteristics 146
Records 146
Record formats 146
Data set organizations 146
Specifying characteristics using the PL/I ENVIRONMENT attribute 147
BKWD 147
CONSECUTIVE 147
CTLASA 148
GENKEY 148
GRAPHIC 150
KEYLENGTH 150
KEYLOC 150
ORGANIZATION 151
RECSIZE 151
REGIONAL(1) 151
SCALARVARYING 152
VSAM 152
Specifying characteristics using DD_ddname environment variables 152
AMTHD 153
APPEND 154
ASA 155
BUFSIZE 155
CHARSET for record I/O 155
CHARSET for stream I/O 155
DELAY 156
DELIMIT 156
LRECL 156
LRMSKIP 156
PROMPT 157
PUTPAGE 157
RECCOUNT 157
RECSIZE 157
RETRY 158
SAMELINE 158
SHARE 159
SKIP0 159
TYPE 159
Associating a PL/I file with a data set 161
Using environment variables 162
Using the TITLE option of the OPEN statement 162
Attempting to use files not associated with data sets 163
How PL/I finds data sets 163
Opening and closing PL/I files 163
Opening a file 163
Closing a file 163
Associating several data sets with one file 164
Combinations of I/O statements, attributes, and options 164
DISPLAY statement input and output 166
PL/I standard files (SYSPRINT and SYSIN) . . . 167
Redirecting standard input, output, and error devices 167

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 two types of data sets—native data sets and workstation VSAM data sets.

- The term native data set is a PL/I term used to define conventional text files and devices associated with the platform in use.
- The term workstation VSAM data set is used to refer to files that are similar to mainframe VSAM data sets. PL/I uses either the DDM or ISAM access method to create and access these types of data sets.

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 LF (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:

/dev/null/
: 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)

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 Line Feed (LF) as characters with special meaning. In particular, the 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 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 two-byte prefix that specifies the number of bytes in the rest of the record. Unlike files with records delimited by 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 191.

Note: Regional in this context means the same thing as REGIONAL(1) does in OS PL/I.

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:

- DDM
- ISAM

DDM access method

DDM data sets are record-oriented files as defined by the Distributed Data Management 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 ISAM local access method and not mainframe ISAM. ISAM data sets are stored in one file and can exist on local file systems only.

Detailed information on workstation VSAM is found in Chapter 13, “Defining and using workstation VSAM data sets,” on page 201.

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 161 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 information
- 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 169.

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-LF file using an editor, because most editors automatically insert LF or similar character sequences.

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.

BKWD

The BKWD option specifies backward processing for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file that is associated with a DDM data set.

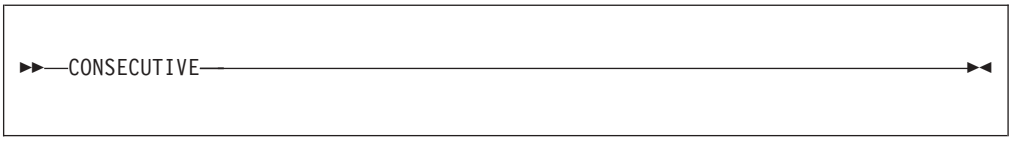
Sequential reads (that is, reads without the KEY option) retrieve the previous record in sequence. For indexed data sets, the previous record is the record with the next lower key.

When a file with the BKWD option is opened, the data set is positioned at the last record. ENDFILE is raised in the normal way when the start of the data set is reached. The BKWD option must not be specified with the GENKEY option.

The WRITE statement is not allowed for files declared with the BKWD option.

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.



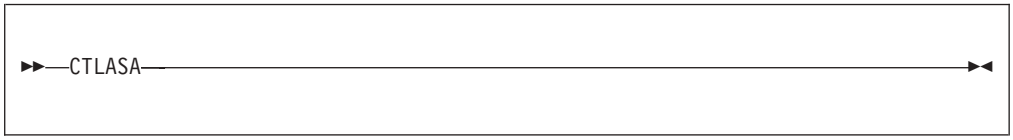
You use the CONSECUTIVE option to access native data sets using either stream-oriented or record-oriented data transmission. You also use it for input files

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 (ANS) print control character. The option applies only to RECORD OUTPUT files associated with consecutive data sets.



The ANS print control characters, listed in Table 14 on page 170, 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 169.

The IBM Proprinter control characters require up to 3 bytes more than the single byte required by an ANS 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 155.

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.

A generic key is a character string that identifies a class of keys; all keys that begin with the string are members of that class. For example, the recorded keys "ABCD", "ABCE", and "ABDF" are all members of the classes identified by the generic keys "A" and "AB", and the first two are also members of the class "ABC"; and the three recorded keys can be considered to be unique members of the classes "ABCD", "ABCE", and "ABDF", respectively.

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an

INDEXED data set from the first nondummy record that has a key in a particular class. You identify the class by including its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although you can retrieve the first record 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 three bytes is assumed:

```
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.

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.

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*.

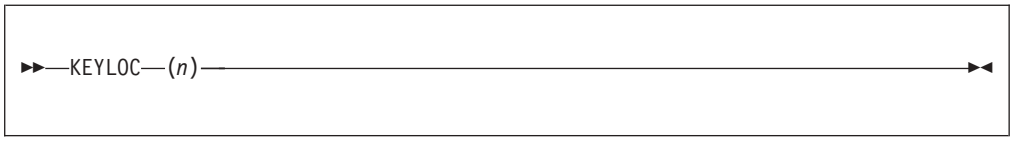
KEYLENGTH

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).

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.

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).



The position, *n*, must be within the limits:

```
1 ≤ n ≤ recordsize - keylength + 1
```

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.

ORGANIZATION

The ORGANIZATION option specifies the organization of the data set associated with the PL/I file.

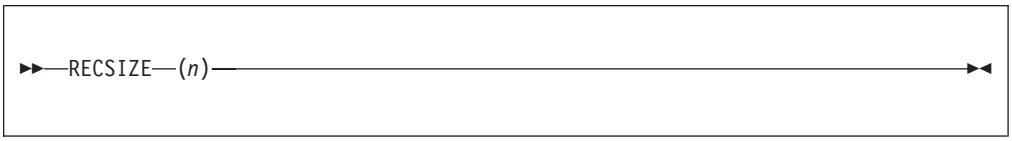
CONSECUTIVE
Specifies that the file is associated with a consecutive data set. A consecutive file may be either a native data set or a workstation VSAM sequential, direct, or keyed data set.

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 nnnn.

RECSIZE

The RECSIZE option specifies the length, *n*, of records in a data set.



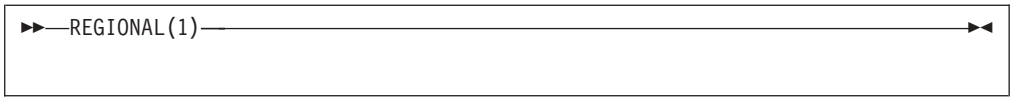
For regional and fixed-length data sets, RECSIZE specifies the length of each record in the data set; for all other data set types, RECSIZE specifies the maximum length records can have.

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.

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 191.

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.

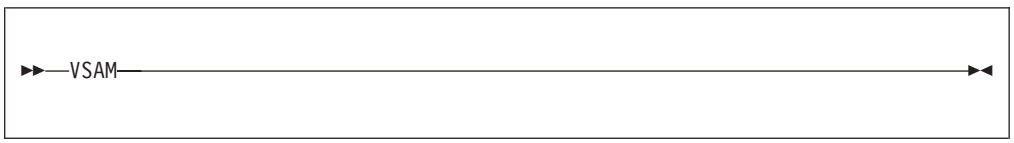
When you specify this option and element VARYING strings are transmitted, you must allow two bytes in the record length to include the length prefix.

A data set created using SCALARVARYING should be accessed only by a file that also specifies SCALARVARYING.

SCALARVARYING and CTLASA must not be specified for the same file, as this causes the first data byte to be ambiguous.

VSAM

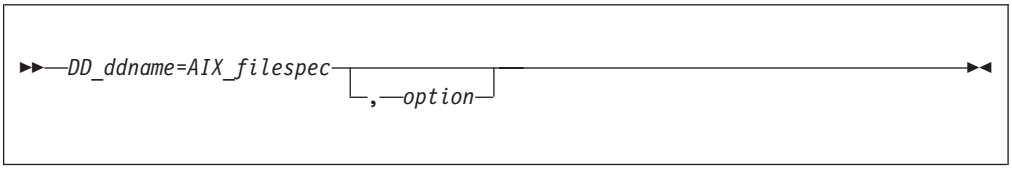
The VSAM option is provided for compatibility with OS PL/I.



Specifying characteristics using DD_ddname environment variables

You use the export command to establish an environment variable that identifies the data set to be associated with a PL/I file, and, optionally, provide additional characteristics of that data set. This information provided by the environment variable is called data definition (or DD) information.

The syntax of the DD_ddname environment variable is:



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 must be in upper case and can be either the name of a file constant or an alternate DDNAME that you specify in the TITLE option of your OPEN statement. The TITLE option is described in "Using the TITLE option of the OPEN statement" on page 162.

For example, an environment variable that associates the file */u/myuser/input.dat* with a PL/I file variable *DD_INPUT* would look like this:

```
DD_INPUT=/u/myuser/input.dat
```

If the file were defined with a record size of 80, the environment variable would look like this:

```
DD_INPUT='/u/myuser/input.dat,recsize(80)'
```

Notice that if the environment variable definition contains parentheses, the complete definition must be enclosed in single quotes.

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.

AIX_filespec
Specifies a file or the name of a device to be associated with the PL/I file. See "Conventional text files and devices" on page 144 for the names you use to specify the appropriate device.

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 159.

AMTHD

The AMTHD option specifies the access method that is to be used to access the data set.

FSYS
Specifies that PL/I is to use its native access methods to access a native file. This is the default.

DDM
Specifies that the DDM access method is to be used to access a DDM file. This is also used if ENCINA files are involved.

Encina SFS file access support

Use the following checklist when accessing ENCINA files from PL/I batch programs:

1. Specify AMTHD(DDM) and the fully-qualified pathname for the file.
2. Make sure SFS PTF U441643 (or later) is applied.
3. Compile and link applications using the `pli_r4` command.
4. Log in to DCE with the DCE principal that has the proper authority to the ENCINA SFS server. This step is unnecessary when the environment variable ENCINA_BINDING_FILE is set.
5. Set up the ENCINA environment variables as follows:
   - `ENCINA_CDS_ROOT` should be set to the name of the registered ENCINA server on which the files are located.
   - `ENCINA_SFS_DATA_VOLUME` should be set to the name of the SFS data volume on which the files are to be created.
   - `ENCINA_SFS_INDEX_VOLUME` should be set to the name of the SFS data volume on which the VSAM alternative index files are to be created.
   - `ENCINA_SFS_VOL` should be set to the name of the SFS data volume.
   - `ENCINA_BINDING_FILE` should be set to the location of the binding translations used when not using DCE servers.
   - `ENCINA_SFS_SERVER` should be set to the name of the registered ENCINA server.

For information regarding ENCINA environment variables, refer to the CICS Administration Reference manual.

The sample `export` statements below show the ENCINA server name as `/.:/encina/sfs/pli`. The volume `sfs_SFS_SERV` is the destination for data and alternate index files.

```
export ENCINA_CDS_ROOT=/.:/encina/sfs/pli
export ENCINA_SFS_DATA_VOLUME=sfs_SFS_SERV
export ENCINA_SFS_INDEX_VOLUME=sfs_SFS_SERV
```

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

AMTHD(DDM) is applied if you specify any of the above options.

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 ANS control character in each record is to be interpreted.

N Specifies that the ANS print control characters are to be translated to IBM Proprinter control characters as records are written to the data set. This is the default.

Y Specifies that the ANS 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 169.

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.

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

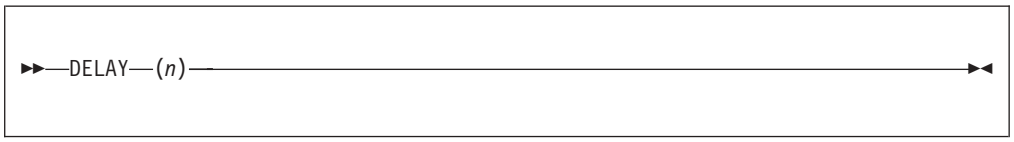
specifying the character set of output files. If you attempt to specify ASIS when using stream I/O, no error is issued and character sets are treated as ASCII.

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.



The default value for DELAY is 0. This option is applicable only to DDM files.

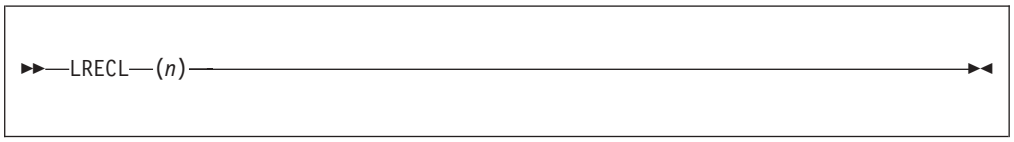
DELIMIT

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.

The sort utility distinguishes text files from binary files with the presence of field delimiters. Input files that contain field delimiters are processed as text files; otherwise, they are considered to be binary files. The library needs this information in order to pass the correct parameters to the sort utility.

LRECL

The LRECL option is the same as the RECSIZE option.



If LRECL is not specified and not implied by a LINESIZE value (except for TYPE(FIXED) files, the default is 1024.

LRMSKIP

The LRMSKIP option allows output to 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.

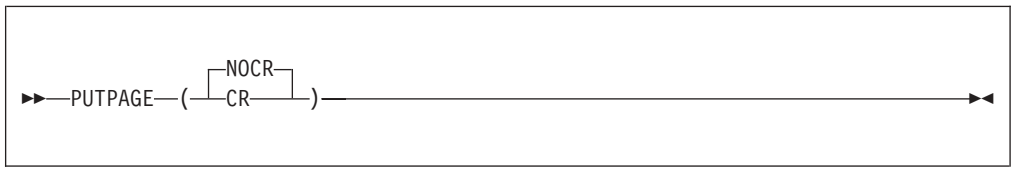
PROMPT

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.

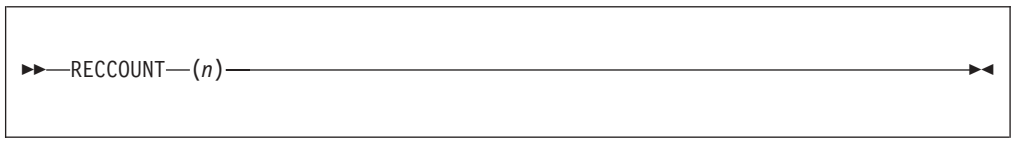


NOCR
Indicates that the form feed character ('0C'x) is not followed by a carriage return character ('0D'x). This is the default.

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.

RECCOUNT

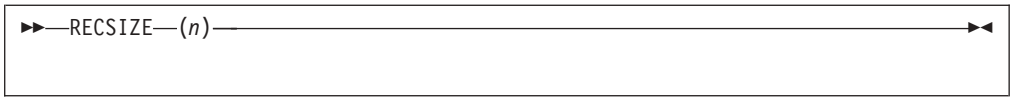
The RECCOUNT option specifies the maximum number of records that can be loaded into a relative or regional data set that is created during the PL/I file opening process.



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.

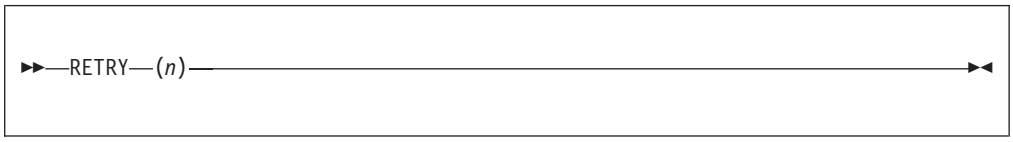


For regional and fixed-length data sets, RECSIZE specifies the length of each record in the data set; for all other data set types, RECSIZE specifies the maximum length records may have.

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.



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.

The following examples show the results of certain combinations of the PROMPT and SAMELINE options:

Example 1

Given the statement `PUT SKIP LIST('ENTER:');`, output is as follows:

prompt(y), sameline(y)	ENTER: (cursor)
prompt(n), sameline(y)	ENTER: (cursor)
prompt(y), sameline(n)	ENTER: (cursor)
prompt(n), sameline(n)	ENTER: (cursor)

Example 2

Given the statement `PUT SKIP LIST('ENTER');`, output is as follows:

prompt(y), sameline(y)	ENTER: (cursor)
prompt(n), sameline(y)	ENTER (cursor)
prompt(y), sameline(n)	ENTER : (cursor)
prompt(n), sameline(n)	ENTER (cursor)

SHARE

The SHARE option specifies the level of file sharing to be allowed.

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.

SKIP0(N)
Specifies that the cursor is to be moved to the beginning of the next line. This is the default.

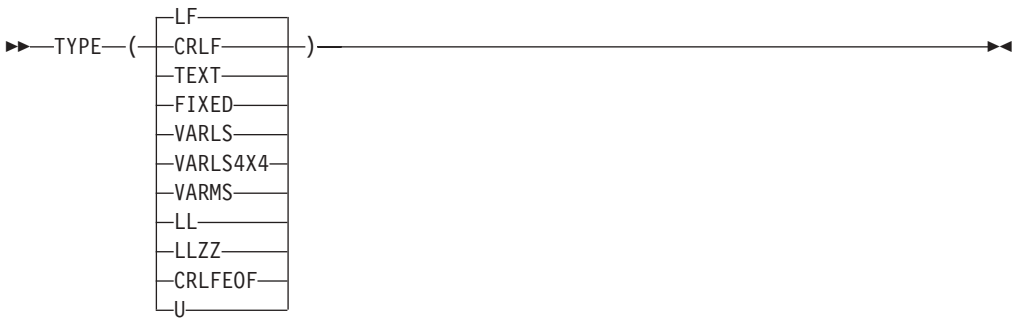
SKIP0(Y)
Specifies that the cursor to be moved to the beginning of the current line.

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:

```
export DD_SYSPRINT='stdout:,SKIP0(Y)'
```

TYPE

The TYPE option specifies the format of records in a native file.



CRLF
Specifies that records are delimited by the CR - LF character combination. ('CR' and 'LF' represent the ASCII values of carriage return and line feed, '0D'x and '0A'x, respectively. See restrictions on 16) For an output file, PL/I places the characters at the end of each record; for an input file, PL/I discards the characters. For both input and output, the characters are not counted in consideration for RECSIZE.

The data set must not contain any record that is longer than the value determined for the record length of the data set.

LF Specifies that records are delimited by the LF character combination. ('LF' represents the ASCII values of feed or '0A'x. See restrictions on 16) For an output file, PL/I places the characters at the end of each record; for an input file, PL/I discards the characters. For both input and output, the characters are not counted in consideration for RECSIZE.

The data set must not contain any record that is longer than the value determined for the record length of the data set.

TEXT
Equivalent to LF.

FIXED
Specifies that each record in the data set has the same length. The length determined for records in the data set is used to recognize record boundaries.

All characters in a TYPE(FIXED) file are considered as data, including control characters if they exist. Make sure the record length you specify reflects the presence of these characters or make sure the record length you specify accounts for all characters in the record.

VARLS
Indicates that records have a two-byte prefix that specifies the number of bytes in the rest of the record and that the length prefix is held in NATIVE format. These records look like NATIVE CHAR VARYING strings.

TYPE(VARLS) datasets 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.

VARLS4X4
Indicates that records have a four-byte prefix and a four-byte suffix. The prefix and suffix each contain the number of bytes in the rest of the record. This number is in NATIVE format and does not include either the four bytes used by the prefix or the four bytes used by the suffix.

Type(VARLS4X4) data sets provide a way to handle FORTRAN sequential unformatted files.

VARMS
Indicates that records have a two-byte prefix that specifies the number of bytes in the rest of the record and that the length prefix is held in NONNATIVE format. These records look like NONNATIVE CHAR VARYING strings.

TYPE(VARMS) data sets provide a way to read SCALARVARYING files downloaded from the mainframe.

LL Indicates that records have a two-byte prefix that specifies the total number of bytes in the record (including the prefix). The length is held in NONNATIVE format.

TYPE(LL) data sets provide a way to read files downloaded from the mainframe with a tool (see VRECGEN.PLI sample program) that appends two bytes.

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 which contain records of variable length and arbitrary byte patterns which cannot be done with 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(LF) is being used, SKIP is replaced by LF 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 export command to establish an environment variable that identifies the data set to be associated with a PL/I file, and, optionally, to specify the characteristics of that data set. The information provided by the environment variable is called data definition (or DD) information.

These environment variable names have the form DD_DDNAME where the *DDNAME* is the name of a PL/I file constant (or an *alternate DDNAME*, as defined below). For example:

```
declare MyFile stream output;

export DD_MYFILE=~/datapath/mydata.dat
```

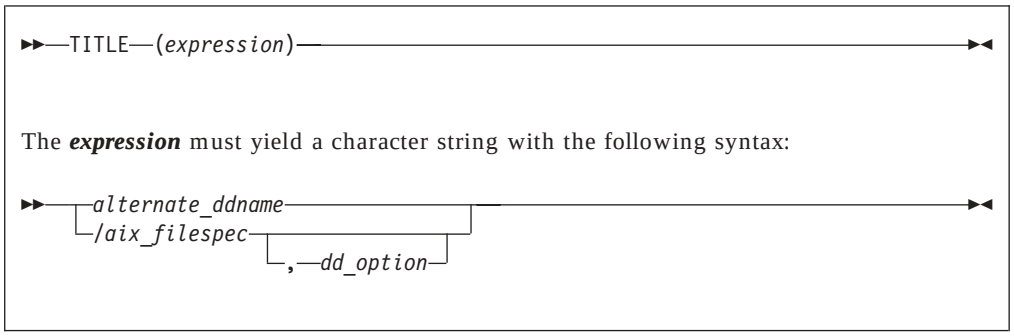
If you are familiar with the IBM mainframe environment, you can think of the environment variable much like you do the:

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 152.

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.



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');
```

aix_filespec

Any valid file specification on the system you are using.

dd_option
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 152.

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 stdin and stdout, respectively.

How PL/I finds data sets

PL/I establishes the path for creating new data sets or accessing existing data sets in one of the following ways:

- The current directory.
- The paths as defined by the `export DD_DDNAME` environment variable.

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:

```
do Ident='A','B','C';
  open file(Master) title('/MASTER1'||Ident||'.DAT');
       .
       .
       .
  close file(Master);
end;
```

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 12 on page 165 lists those for native data sets and Table 13 on page 165 lists those for workstation VSAM data sets.

Table 12. Statements, attributes, and options for native data sets

Statements	File attributes	ENVIRONMENT options	DD_DDNAME options
PUT	ENVIRONMENT FILE OUTPUT PRINT STREAM	CONSECUTIVE GRAPHIC RECSIZE(n)	AMTHD(FSYS) APPEND(Y | N) ASA(Y | N) file_spec RECSIZE(n) SHARE(NONE | READ | ALL) TERMLBUF(n) TYPE(CRLF | TEXT | FIXED)
GET	ENVIRONMENT FILE STREAM INPUT	CONSECUTIVE GRAPHIC RECSIZE(n)	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE | READ | ALL) TERMLBUF(n) TYPE(CRLF | TEXT | FIXED)
WRITE	BUFFERED | UNBUFFERED DIRECT | SEQUENTIAL ENVIRONMENT FILE KEYED RECORD OUTPUT | UPDATE	CONSECUTIVE | REGIONAL(1) CTLASA RECSIZE(n) SCALARVARYING	AMTHD(FSYS) APPEND(Y | N) file_spec RECSIZE(n) SHARE(NONE | READ | ALL) TERMLBUF(n) TYPE(CRLF | TEXT | FIXED)
LOCATE	BUFFERED ENVIRONMENT FILE KEYED RECORD OUTPUT SEQUENTIAL	CONSECUTIVE | REGIONAL(1) CTLASA RECSIZE(n)	AMTHD(FSYS) APPEND(Y | N) file_spec RECSIZE(n) SHARE(NONE | READ | ALL) TYPE(CRLF | TEXT | FIXED)
READ	BUFFERED | UNBUFFERED DIRECT | SEQUENTIAL ENVIRONMENT FILE INPUT | UPDATE KEYED RECORD	CONSECUTIVE | REGIONAL(1) RECSIZE(n) SCALARVARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE | READ | ALL) TERMLBUF(n) TYPE(CRLF | TEXT | FIXED)
REWRITE	BUFFERED | UNBUFFERED DIRECT | SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	CONSECUTIVE | REGIONAL(1) RECSIZE(n) SCALARVARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE | READ | ALL) TYPE(CRLF | TEXT | FIXED)
DELETE	BUFFERED | UNBUFFERED DIRECT | SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	REGIONAL(1) RECSIZE(n) SCALARVARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE | READ | ALL)

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 13. Statements, attributes, and options for workstation VSAM data sets

Statements	File attributes	ENVIRONMENT options	DD_DDNAME options
PUT	ENVIRONMENT FILE OUTPUT PRINT STREAM	ORGANIZATION(CONSECUTIVE) GRAPHIC RECSIZE(n)	AMTHD(DDM) APPEND(Y | N) ASA(Y | N) file_spec RECSIZE(n) SHARE(NONE | READ | ALL)

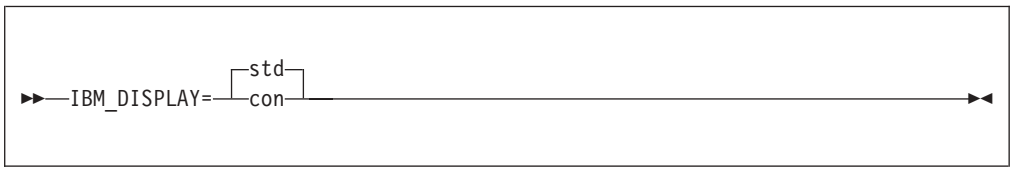
Table 13. Statements, attributes, and options for workstation VSAM data sets (continued)

Statements	File attributes	ENVIRONMENT options	DD_DDNAME options
GET	ENVIRONMENT FILE STREAM INPUT	ORGANIZATION(CONSECUTIVE) GRAPHIC RECSIZE(n)	AMTHD(DDM|) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
WRITE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE KEYED RECORD OUTPUT|UPDATE	ORGANIZATION|VSAM CTLASA RECSIZE(n) SCALARVARYING	AMTHD(DDM|) ASA(Y|N) APPEND(Y|N) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
LOCATE	BUFFERED ENVIRONMENT FILE KEYED RECORD OUTPUT SEQUENTIAL	ORGANIZATION|VSAM CTLASA RECSIZE(n) SCALARVARYING	AMTHD(DDM|) APPEND(Y|N) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
READ	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE INPUT|UPDATE KEYED RECORD	ORGANIZATION|VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDM|) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
REWRITE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	ORGANIZATION|VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDM) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
DELETE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	ORGANIZATION|VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDM) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)

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:



std
Specifies that the DISPLAY statement is to be associated with standard output. This is the default.

con
Specifies that the DISPLAY statement is to be associated with the /dev/tty file.

You can redirect display statements to a file, for example:

```
Hello1: proc options(main);
  display('Hello!');
end;
```

After compiling and linking the program, you could invoke it from the command line by entering:

```
hello1 > 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)

SYSIN is read from stdin and SYSPRINT is directed to stdout by default. If you want either to be associated differently, you must use the TITLE option of the OPEN statement, or establish a DD_DDNAME environment variable naming a data set or another device.

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:

```
Hello2: proc options(main);
  put list('Hello!');
end;
```

After compiling and linking the program, you could invoke it from the command line by entering:

```
hello2 > 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

Printer-destined files 169
Using stream-oriented data transmission 170
Defining files using stream I/O 171
ENVIRONMENT options for stream-oriented data transmission 171
Creating a data set with stream I/O 171
Essential information 171
Example 172
Accessing a data set with stream I/O 173
Essential information 174
Example 174
Using PRINT files 175
Controlling printed line length 176
Overriding the tab control table 178
Using SYSIN and SYSPRINT files 179
Controlling input from the console 180
Using files conversationally 181
Format of data 181
Stream and record files 181
Capital and lowercase letters 182
End of file 182
Controlling output to the console. 182
Format of PRINT files 182
Stream and record files 182
Example of an interactive program 182
Using record-oriented I/O 183
Defining files using record I/O 184
ENVIRONMENT options for record-oriented data transmission 185
Creating a data set with record I/O 185
Essential information 185
Accessing and updating a data set with record I/O. 185
Essential information 186
Examples of consecutive data sets 186

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 DDM data set.

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 (ANS) carriage control character (see Table 14 on page 170).

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 14. ANS print control characters

Character	Meaning
(blank)	Skip 1 line before printing
0	Skip 2 lines before printing
hyphen (-)	Skip 3 lines before printing
+	Do not skip any lines before printing
1	Skip to next page before printing
2	Skip 3 lines before printing
3	Skip 3 lines before printing
4	Skip 3 lines before printing
5	Skip 3 lines before printing
6	Skip 3 lines before printing
7	Skip 3 lines before printing
8	Skip 3 lines before printing
9	Skip 3 lines before printing
A	Skip 3 lines before printing
B	Skip 3 lines before printing
C	Skip 3 lines before printing

The translation to IBM Proprinter control characters is as follows:

Table 15. IBM Proprinter equivalents to ANS control characters

ANS Character	Proprinter Characters (in hexadecimal)
(blank)	0A
0	0A 0A
-	0A 0A 0A
+	0D
1	0C
2 to 9, A to C	0A 0A 0A

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. 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, which allows your PL/I program to ignore record boundaries and to treat a data set as a continuous stream of data values. Data values are either in character format or graphic format—that is, in DBCS (double byte character set) form. 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 DBCS characters 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 DBCS data (graphics). DBCS data can be entered, displayed and printed if the appropriate devices have DBCS support. 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:

```
declare
  Filename file stream
           input | {output [print]}
           environment(options);
```

The FILE attribute is described in the *PL/I Language Reference*. The PRINT attribute is described further in "Using PRINT files" on page 175.

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 147.

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 162 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 178).
- The associated data set is the terminal (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:

- 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 173 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 `export 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 `export DD_WORK=BDAY.OCT` should be used to associate the consecutive output file WORK with the disk data set BDAY.OCT.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Create a CONSECUTIVE data set with 30-byte records containing */
/*    names and birthdays of people whose birthdays are in October. */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_WORK='BDAY.OCT'                                    */
/*      export DD_SYSIN='BDAY.INP,RECSIZE(80)'                       */
/*********************************************************************/

BDAY: proc options(main);

  dcl Work file stream output,
      1 Rec,
          3 Name char(19),
          3 BMonth char(3),
          3 Pad1 char(1),
          3 BDate char(2),
          3 Pad2 char(1),
          3 BYear char(4);

   dcl Eof bit(1) init('0'b);
   dcl In char(30) def Rec;

   on endfile(sysin) Eof='1'b;

   open file(Work) linesize(400);
   get file(sysin) edit(In)(a(30));
   do while (¬Eof);
    if BMonth = 'OCT'
       then put file(Work) edit(In)(a(30));
       else;
    get file(sysin) edit(In)(a(30));
   end;
   close file(Work);
end BDAY;


BDAY.INP contains the input data used at execution time:

LUCY   D.          MAR 15 1950
REGINA W.          OCT 09 1971
GARY   M.          DEC 01 1964
PETER  T.          MAY 03 1948
JANE   K.          OCT 24 1939
```

Figure 8. Creating a data set with stream-oriented data transmission

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 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 171.

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 (stdout:, or stderr:), value = 120.

Example

The program in Figure 9 on page 175 reads the data created by the program in Figure 8 on page 173 and uses the data set SYSPRINT to display that data. The SYSPRINT data set is associated with the stdout 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 179.)

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Read a CONSECUTIVE data set and print the 30-byte records      */
/*    to the screen.                                                 */
/*                                                                   */
/*  USAGE                                                            */
/*    The following command is required to establish                 */
/*    the environment variable to run this program:                  */
/*                                                                   */
/*      export DD_WORK='BDAY.OCT'                                    */
/*                                                                   */
/*    Note: This sample program uses the CONSECUTIVE data set        */
/*          created by the previous sample program BDAY.             */
/*                                                                   */
/*********************************************************************/

BDAY1: proc options(main);

  dcl Work file stream input;

  dcl Eof bit(1) init('0'b);

  dcl In char(30);

  on endfile(Work) Eof='1'b;

  open file(Work);
  get file(Work) edit(In)(a(30));
  do while (¬Eof);
   put file(sysprint) skip edit(In)(a);
   get file(Work) edit(In)(a(30));
  end;
  close file(Work);
end BDAY1;
```

Figure 9. Accessing a data set with stream-oriented data transmission

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 169).

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 177 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’.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Create a SEQUENTIAL data set.                                  */
/*                                                                   */
/*  USAGE                                                            */
/*    The following command is required to establish                 */
/*    the environment variable to run this program:                  */
/*                                                                   */
/*      export DD_TABLE='MYTAB.DAT,ASA(Y)'                           */
/*                                                                   */
/*********************************************************************/

SINE: proc options(main);

 /*  Build a table of SINE values.                */
  dcl Table       file stream output print;
  dcl Deg         fixed dec(5,1) init(0);  /* init(0) for endpage  */
  dcl Min         fixed dec(3,1);
  dcl PgNo        fixed dec(2)   init(0);
  dcl Oncode      builtin;
  dcl I           fixed dec(2);

  on error
    begin;
      on error system;
      display ('oncode = '|| Oncode);
    end;
```

Figure 10. Creating a print file via stream data transmission (Part 1 of 2). (The example in Figure 15 on page 190 prints this file)

```
   on endpage(Table)
     begin;
       if PgNo ¬= 0 then
         put file(Table) edit ('page',PgNo)
             (line(55),col(80),a,f(3));
       if Deg ¬= 360 then
         do;
           put file(Table) page edit ('Natural Sines') (a);

           put file(Table) edit ((I do I = 0 to 54 by 6))
                                (skip(3),10 f(9));
           PgNo = PgNo + 1;
         end;
       else
         put file(Table) page;
     end;

   open file(Table) pagesize(52) linesize(102);
   signal endpage(Table);

   put file(Table) edit
    ((Deg,(sind(Deg+Min) do Min = 0 to .9 by .1) do Deg = 0 to 359))
    (skip(2), 5 (col(1), f(3), 10 f(9,4) ));
   put file(Table) skip(52);
 end SINE;
```

Figure 10. Creating a print file via stream data transmission (Part 2 of 2). (The example in Figure 15 on page 190 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.

The program in Figure 15 on page 190 uses record-oriented data transmission to print the table created by the program in Figure 10.

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)) native 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—Tab*n*:
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.

```
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)) native 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:

```
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 console

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 console device.

You can be prompted for input to stream files by a colon (:) if you specify PROMPT(Y), see “PROMPT” on page 157. The colon is visible each time a GET statement is executed in the program. If you enter a line that does not contain enough data to complete execution of the GET statement, a further prompt is displayed. The GET statement causes the system to go to the next line. You can then enter the required data.

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 another line. 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:

```
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 console 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:

  ```
  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
  ```

  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:

  ```
  get edit(Name)(a(15));
  ```

  You could enter these five characters followed immediately by the ENTER.

  ```
  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 mark and raise the ENDFILE condition.

Controlling output to the console

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 (stdout:, 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 178. For other than standard layout, use the information about PLITABS provided in Figure 11 on page 178. 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 183 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.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Create a SEQUENTIAL data set using a console dialog.           */
/*                                                                   */
/*  USAGE                                                            */
/*    The following command is required to establish                 */
/*    the environment variable to run this program:                  */
/*                                                                   */
/*      export DD_PHONES='INT1.DAT,APPEND(Y)'                        */
/*                                                                   */
/*********************************************************************/

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 16 on page 184 lists the data transmission statements and options that you can use to create and access a consecutive data set 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 16. Statements and options allowed for creating and accessing consecutive data sets

File Declaration[1]	Valid Statements,[2] with Options You Must Specify	Other Options you can Specify
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference); LOCATE based-variable FILE(file-reference);	SET(pointer reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INPUT(reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	

Notes:

[1] The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT

[2] The statement READ FILE (file-reference); is a valid statement and is equivalent to READ FILE(file-reference) IGNORE (1);

Defining files using record I/O

You define files for record-oriented data transmission by using a file declaration with the following attributes:

```
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 147.

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 16 on page 184 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 163).
- 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 (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 187.

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.

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 16 on page 184 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 187. 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.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Merge 2 data sets creating a CONSECUTIVE data set.             */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_OUT='CON4.DAT'                                     */
/*      export DD_INPUT1='EVENS.INP'                                 */
/*      export DD_INPUT2='ODDS.INP'                                  */
/*                                                                   */
/*********************************************************************/

 MERGE: proc options(main);

   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 (Part 1 of 3)

```
    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);

  end MERGE;
```

Figure 14. Merge Sort—Creating and accessing a consecutive data set (Part 2 of 3)

```
Here is a sample of EVENS.INP:

BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ

Here is a sample of ODDS.INP:

AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
KKKKKK
```

Figure 14. Merge Sort—Creating and accessing a consecutive data set (Part 3 of 3)

The program in Figure 15 uses record-oriented data transmission to print the table created by the program in Figure 10 on page 177.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Print a SEQUENTIAL data set created by the SINE program.       */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_TABLE='MYTAB.DAT'                                  */
/*      export DD_PRINTER='PRN'                                      */
/*                                                                   */
/*********************************************************************/

PRT: proc options(main);

  dcl Table       file record input sequential;
  dcl Printer     file record output seql
                       env(recsize(200) ctlasa);
  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(Printer);

  read file(Table) into(Line);             /* priming read         */

  do while (Table_Eof = False);
    if Line='' then                        /* insert blank lines   */
       Line= ' ';
    write file(Printer) from(Line);
    read file(Table) into(Line);
  end;

  close file(Table),
        file(Printer);
end PRT;
```

Figure 15. Printing record-oriented data transmission

Chapter 12. Defining and using regional data sets

Defining files for a regional data set 193
Specifying ENVIRONMENT options 193
Essential information for creating and accessing regional data sets 194
Using keys with regional data sets 194
Using REGIONAL(1) data sets 194
Dummy records 194
Creating a REGIONAL(1) data set 195
Example 195
Accessing and updating a REGIONAL(1) data set 197
Sequential access 197
Direct access. 198
Example 198

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 17 for a list of the data transmission statements and options that you can use to create and access a REGIONAL(1) data set.

Table 17. Statements and options allowed for creating and accessing regional data sets

File Declaration[1]	Valid Statements,[2] With Options You Must Include	Other Options You Can Also Include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); LOCATE based-variable FROM(file-reference) KEYFROM(expression);	SET(pointer-reference)

Table 17. Statements and options allowed for creating and accessing regional data sets (continued)

File Declaration[1]	Valid Statements,[2] With Options You Must Include	Other Options You Can Also Include
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE[3] BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference) FROM(reference);	
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	

Table 17. Statements and options allowed for creating and accessing regional data sets (continued)

File Declaration[1]	Valid Statements,[2] With Options You Must Include	Other Options You Can Also Include
DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); DELETE FILE(file-reference) KEY(expression);	

Notes:

[1] The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED.

[2] The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);

[3] 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:

```
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:

```
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:

REGIONAL(1)
RECSIZE

SCALARVARYING

These options are described in “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 147.

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:

- 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 163).
- 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 X’FF’ 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 17 on page 191 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.

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 on page 196. 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.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Create a REGIONAL(1) data set.                                 */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*     export DD_SYSIN='CRG.INP,RECSIZE(30)'                         */
/*     export 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 (Part 1 of 2)

```
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.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 16. Creating a REGIONAL(1) data set (Part 2 of 2)

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 17 on page 191 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 185.

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 199. 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.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Update a REGIONAL(1) data set.                                 */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_SYSIN='ACR.INP,RECSIZE(30)'                        */
/*      export DD_NOS='NOS.DAT,APPEND(Y)'                            */
/*                                                                   */
/*    Note: This sample program is using the regional dataset,       */
/*          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 (Part 1 of 3)

```
    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);
    close file(Sysin),file(Nos);
    put file(sysprint) page;
    open file(Nos) sequential input;
    on endfile (Nos) nos_Eof = '1'b;
    read file(Nos) into(IoField) keyto(CNewNo);
    do while(¬Nos_Eof);
       if unspec(Byte) ¬= (8)'1'b then
           put file(sysprint) skip
               edit (CNewNo,' ',IoField)(a(2),a(1),a);
       read file(Nos) into(IoField) keyto(CNewNo);
    end;
   close file(Nos);
   end ACR1;

    end;
```

Figure 17. Updating a REGIONAL(1) data set (Part 2 of 3)

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
```

Figure 17. Updating a REGIONAL(1) data set (Part 3 of 3)

Chapter 13. Defining and using workstation VSAM data sets

Remote file access 201
Workstation VSAM organization 202
Creating and accessing workstation VSAM data sets. 202
Determining which type of workstation VSAM data set you need 202
Accessing records in workstation VSAM data sets. 202
Using keys for workstation VSAM data sets . . 203
Using keys for workstation VSAM keyed data sets 203
Using sequential record values 204
Using relative record numbers. 204
Choosing a data set type 204
Defining files for workstation VSAM data sets . . 204
Specifying options of the PL/I ENVIRONMENT attribute 205
Adapting existing programs for workstation VSAM. 205
Adapting programs using CONSECUTIVE files 206
Adapting programs using INDEXED files 206
Adapting programs using REGIONAL(1) files 206
Adapting programs using VSAM files . . . 206
Using workstation VSAM sequential data sets . . 208
Using a sequential file to access a workstation VSAM sequential data set 209
Defining and loading a workstation VSAM sequential data set. 209
Updating a sequential data set 210
Workstation VSAM keyed data sets 211
Loading a workstation VSAM keyed data set 213
Using a SEQUENTIAL file to access a workstation VSAM keyed data set 215
Using a DIRECT file to access a workstation VSAM keyed data set 215
Workstation VSAM direct data sets 218
Loading a workstation VSAM direct data set 220
Using a SEQUENTIAL file to access a workstation VSAM direct data set 222
Using READ statements. 222
Using WRITE statements 222
Using the REWRITE or DELETE statements 223
Using a DIRECT file to access a workstation VSAM direct data set. 223

This chapter describes how you use Virtual Storage Access Method (VSAM) data sets on your workstation—Distributed Data Management (DDM) data sets—for record-oriented data transmission.

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.

Remote file access

DDM data sets can be created either locally or on remote systems. .

The term local implies either local devices or devices on the LAN, such as a LAN server. A remote system implies a mainframe system such as MVS/ESA or OS/400. For information on how to use remote file access or use the Structured File Server Adaptor for AIX (SFS), see *SMARTdata UTILITIES for AIX VSAM in a Distributed Environment*, SC26-7064.

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.

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 18 on page 203 shows how data could be stored in the three different types of workstation VSAM data sets and illustrates their respective advantages and disadvantages.

Table 18. Types and advantages of workstation VSAM data sets

Data Set Type	Method of Loading	Method of Reading	Method of Updating	Pros and Cons
Sequential	Sequentially (forward only) The sequential record value of each record can be obtained and used as a key	SEQUENTIAL backward or forward KEYED using the sequential record value Positioning by key followed by sequential either backward or forward	New records at end only Access can be sequential or KEYED Record deletion allowed	***Advantages*** Simple fast creation ***Uses*** For uses where data is primarily accessed sequentially
Keyed	Either sequentially or randomly by key	KEYED by specifying key of record SEQUENTIAL backward or forward in order of any index Positioning by key followed by sequential reading either backward or forward	KEYED specifying a key SEQUENTIAL following positioning by key Record deletion allowed Record insertion allowed	***Advantages*** Complete access and updating ***Uses*** For uses where access is related to key
Direct	Sequentially starting from slot 1 KEYED specifying number of slot Positioning by key followed by sequential writes	KEYED specifying numbers as key Sequential forward or backward omitting empty records	Sequentially starting at a specified slot and continuing with next slot Keyed specifying numbers as key Record deletion allowed Record insertion into empty slots allowed	***Advantages*** Speedy access to record by number ***Disadvantages*** Structure tied to numbering sequences ***Uses*** For use where records are accessed by number

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.

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 available to you. Workstation VSAM data sets can provide all the function of the other types of data sets, plus additional function 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 function.

Table 18 on page 203 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.

Table 19 on page 208, Table 20 on page 211, and Table 21 on page 218 show the statements allowed for sequential data sets, keyed data sets, and direct data sets, respectively.

Defining files for workstation VSAM data sets

You define a workstation VSAM sequential data set by using a file declaration with the following attributes:

```
dcl Filename file record
             input | output | update
             sequential
             buffered
            [keyed]
             environment(organization(consecutive));
```

You define a workstation VSAM keyed data set by using a file declaration with the following attributes:

```
dcl Filename file record
             input | output | update
             sequential | direct
             buffered | unbuffered
            [keyed]
             environment(organization(indexed));
```

You define a workstation VSAM direct data set by using a file declaration with the following attributes:

```
dcl Filename file record
             input | output | update
             direct | sequential
             unbuffered | buffered
            [keyed]
             environment(organization(relative));
```

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:

```
BKWD
CONSECUTIVE
CTLASA
GENKEY
GRAPHIC
KEYLENGTH
KEYLOC
ORGANIZATION(CONSECUTIVE|INDEXED|RELATIVE)
RECSIZE
SCALARVARYING
VSAM
```

For a complete explanation of these ENVIRONMENT options and how to use them, see “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 147. 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 152.

Adapting existing programs for workstation VSAM

This section is intended primarily for OS PL/I 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.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.

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 18 on page 207 is an example of creating a workstation VSAM keyed data set.

```
/*********************************************************************/
/*                                                                   */
/*  NAME - ISAM0.PLI                                                 */
/*                                                                   */
/*  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 18. 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 19.

Table 19. Statements and options allowed for loading and accessing workstation VSAM sequential data sets

File declaration[1]	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference);	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression)[3] or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference)[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-eference);	FROM(reference) and/or KEY(expression)[3]
	DELETE FILE(file-reference);	KEY(expression)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression)[3] or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	KEY(expression)[3]

Table 19. Statements and options allowed for loading and accessing workstation VSAM sequential data sets (continued)

File declaration[1]	Valid statements, with options you must include	Other options you can also include

Notes:

[1] The complete file declaration would include the attributes FILE, 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 19 on page 210 is an example of a program that defines and loads a workstation VSAM sequential data set.

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:

```
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.

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Define and load an ISAM sequential data set.                   */
/*                                                                   */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_IN='ISAM1.INP,RECSIZE(38)'                         */
/*      export DD_FAMFILE='ISAM1.OUT,AMTHD(ISAM),RECSIZE(38)'        */
/*                                                                   */
/*********************************************************************/

 CREATE: proc options(main);

    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);
  end CREATE;

The input data for this program might look like this:

Fred                    69          M
Andy                    70          M
Susan                   72          F
```

Figure 19. Defining and loading a workstation VSAM sequential data set

Updating a sequential data set

The program illustrated in Figure 19 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 20.

Table 20. Statements and options allowed for loading and accessing workstation VSAM keyed data sets

File declaration[1]	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	SET(pointer-reference)
	LOCATE based-variable FILE(file-reference) KEYFROM(expression);	
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference)	KEY(expression)

Table 20. Statements and options allowed for loading and accessing workstation VSAM keyed data sets (continued)

File declaration[1]	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	REWRITE FILE(file-reference) FROM(reference);	KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
DIRECT[3] INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT[3] INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT[3] UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	DELETE FILE(file-reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Table 20. Statements and options allowed for loading and accessing workstation VSAM keyed data sets (continued)

File declaration[1]	Valid statements, with options you must include	Other options you can also include
DIRECT[3] UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); DELETE FILE(file-reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Notes:

[1] The complete file declaration 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 20 on page 214 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.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Load an ISAM keyed data set.                                   */
/*                                                                   */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_DIREC='ISAM2.OUT,AMTHD(ISAM)'                      */
/*      export DD_SYSIN='ISAM2.INP,RECSIZE(80)'                      */
/*                                                                   */
/*********************************************************************/

NAMELD: proc options(main);

  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);
end NAMELD;
```

Figure 20. Defining and loading a workstation VSAM keyed data set (Part 1 of 2)

```
The input file for this program could be:

ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESMAN,D.         141
CORY,G.             336
ELLIOTT,D.          875
FIGGINS,S.          413
HARVEY,C.D.W.       205
HASTINGS,G.M.       391
KENDALL,J.G.        294
LANCASTER,W.R.      624
MILES,R.            233
NEWMAN,M.W.         450
PITT,W.H.           515
ROLF,D.E.           114
SHEERS,C.D.         241
SURCLIFFE,M.        472
TAYLOR,G.C.         407
WILTON,L.W.         404
WINSTONE,E.M.       307
```

Figure 20. Defining and loading a workstation VSAM keyed data set (Part 2 of 2)

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 (or in descending key order if you use the BKWD option). 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.

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 21 shows one method you can use to update a keyed data set.

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Update an ISAM keyed data set by key.                          */
/*                                                                   */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_DIREC='ISAM2.OUT,AMTHD(ISAM)'                      */
/*      export 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);

  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)
  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 21. Updating a workstation VSAM keyed data set (Part 1 of 2)

```
   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);
 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.       183A
CORY,G.                D
READ,K.M.           001A
PITT,W.H.              X
ROLF,D.E.              D
ELLIOTT,D.          291C
HASTINGS,G.M.          D
BRAMLEY,O.H.        439C
```

Figure 21. Updating a workstation VSAM keyed data set (Part 2 of 2)

A DIRECT update file is used and the data is altered according to a code that is passed in the records in the file SYSIN:

A Add a new record
C Change the number of an existing name
D Delete a record

The name, number, and code are read in and action taken according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished 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 21. Statements and options allowed for loading and accessing workstation VSAM direct data sets

File declaration[1]	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); [2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-expression); [2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)

Table 21. Statements and options allowed for loading and accessing workstation VSAM direct data sets (continued)

File declaration[1]	Valid statements, with options you must include	Other options you can also include
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT INPUT UNBUFFERED	READ FILE(file-reference) KEY(expression);	
DIRECT UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); READ FILE(file-reference) SET(pointer-reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); DELETE FILE(file-reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); DELETE FILE(file-reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Table 21. Statements and options allowed for loading and accessing workstation VSAM direct data sets (continued)

File declaration[1]	Valid statements, with options you must include	Other options you can also include

Notes:

[1] The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, 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 203).

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 22 on page 221 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.

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.

```
/*********************************************************************/
/*  DESCRIPTION                                                      */
/*    Load an ISAM direct data set.                                  */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      export DD_SYSIN='ISAM4.INP,RECSIZE(80)'                      */
/*      export DD_NOS='ISAM4.OUT,AMTHD(ISAM),RECCOUNT(100)'          */
/*********************************************************************/
 CREATD: proc options(main);

   dcl  Nos file record output direct keyed
        env(organization(relative) recsize(20) );

   dcl  Sysin file input record;
   dcl 1   In_Area,
        2   Name   char(20),
        2   Number char( 2);
   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);
  end CREATD;
```

Figure 22. Loading a workstation VSAM direct data set (Part 1 of 2)

```
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 22. Loading a workstation VSAM direct data set (Part 2 of 2)

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 dataset.

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 a you were using a KEYED SEQUENTIAL file.

Figure 23 on page 224 shows a direct data set being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under workstation VSAM.

In the second half of the program, the updated file is printed. Again, there is no need to check for the empty records as there is in REGIONAL(1).

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Update an ISAM direct data set by key.                         */
/*                                                                   */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program.                 */
/*                                                                   */
/*      export DD_SYSIN='ISAM5.INP,RECSIZE(80)'                      */
/*      export DD_NOS='ISAM4.OUT,AMTHD(ISAM),APPEND(Y)'              */
/*                                                                   */
/*  Note: This sample program is using the direct ISAM dataset       */
/*        ISAM4.OUT created by the previous sample program CREATD.   */
/*                                                                   */
/*********************************************************************/

 UPDATD: proc options(main);

   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 23. Updating a workstation VSAM direct data set by key (Part 1 of 3)

```
/* 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;
```

Figure 23. Updating a workstation VSAM direct data set by key (Part 2 of 3)

```
    read file(Nos) into(IoField) keyto(CNewNo);
    do while(¬Nos_Eof);
       put file (sysprint) skip
           edit (CNewNo,IoField)(a(5),a);
       read file(Nos) into(IoField) keyto(CNewNo);
    end;
   close file(Nos);
   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.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
```

Figure 23. Updating a workstation VSAM direct data set by key (Part 3 of 3)

Part 5. Advanced topics

Chapter 14. Using user exits

Using the compiler user exit 229
Procedures performed by the compiler user exit 229
Activating the compiler user exit 230
The IBM-supplied compiler exit, IBMUEXIT . . 230
Customizing the compiler user exit 231
Modifying IBMUEXIT.INF 231
Writing your own compiler exit 232
Structure of global control blocks. 232
Writing the initialization procedure 233
Writing the message filtering procedure . . 233
Writing the termination procedure 234
Using data conversion tables 234

PL/I provides a number of user exits that allow you to customize the PL/I product to suit your needs. The 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.

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 24, 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.

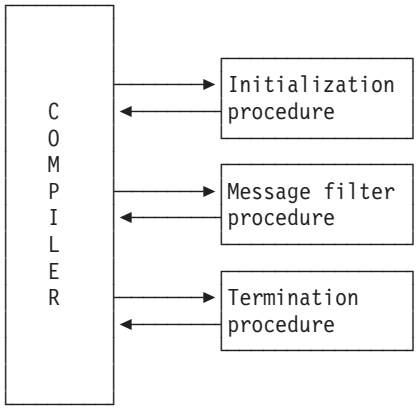


Figure 24. PL/I compiler user exit procedures

Each of the three procedures is passed two different control blocks:

- A *global control block* that contains information about the compilation. This is passed as the first parameter. For specific information on the global control block, see "Structure of global control blocks" on page 232.
- 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 233, "Writing the message filtering procedure" on page 233, and "Writing the termination procedure" on page 234.

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 54.

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 231) 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.

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 232. Please refer to the field "Uex_UIB_User_char_str" in the section "Structure of global control blocks" on page 232 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
Executable load module for IBMUEXIT.PLI that can be FETCHed. In order to build this file, issue the following commands from the AIX command line:

```
pli -e ibmuexit
```

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.

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 25.

```
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-------------------------------
  'IBM'     1041       -1          1      Comment spans multiple lines
  'IBM'     1044       -1          1      FIXED BIN 7 mapped to 1 byte
  'IBM'     1172        0          0      Select without OTHERWISE
  'IBM'     1052       -1          1      Nodescriptor with * extent args
  'IBM'     1047       12          0      Reorder inhibits optimization
  'IBM'     8009       -1          1      Semicolon in string constant
  'IBM'     1107       12          0      Undeclared ENTRY
  'IBM'     1169        0          1      Precision of result determined by arg
```

Figure 25. 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.
- 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.

```
Dcl
  1 Uex_UIB           native based( null() ),
    2 Uex_UIB_Length         fixed bin(31),

    2 Uex_UIB_Exit_token     pointer,        /* for user exit's use */

    2 Uex_UIB_User_char_str  pointer,        /* to exit option str  */
    2 Uex_UIB_User_char_len  fixed bin(31),

    2 Uex_UIB_Filename_str   pointer,        /* to source filename  */
    2 Uex_UIB_Filename_len   fixed bin(31),

    2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs   */
    2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs   */

    2 Uex_UIB_Exit_Routs,                    /* exit entries set at
                                                initialization      */
      3 ( Uex_UIB_Termination,
          Uex_UIB_Message_Filter,            /* call for each msg   */
          *, *, *, * )
        limited entry (
             *,                             /* to Uex_UIB          */
             *,                             /* to a request area   */
         );
```

Data Entry Fields

- **Uex_UIB_ Length**: Contains the length of the control block in bytes. The value is `storage (Uex_UIB)`.
- **Uex_UIB_Exit_token**: Used by the user exit procedure. For example, the initialization may set it to a data structure which is used by both the message filter, and the termination procedures.
- **Uex_UIB_User_char_str**: Points to an optional character string, if you specify it. For example, in `pli filename (EXIT ('string'))...fn` can be a character string up to thirty-one characters in length.
- **Uex_UIB_char_len**: Contains the length of the string pointed to by the `User_char_str`. The compiler sets this value.
- **Uex_UIB_Filename_str**: Contains the name of the source file that you are compiling, and includes the drive and subdirectories as well as the filename. The compiler sets this value.
- **Uex_UIB_Filename_len**: Contains the length of the name of the source file pointed to by the `Filename_str`. The compiler sets this value.
- **Uex_UIB_return_code**: Contains the return code from the user exit procedure. The user sets this value.
- **Uex__UIB_reason_code**: Contains the procedure reason code. The user sets this value.
- **Uex_UIB_Exit_Routs**: Contains the exit entries set up by the initialization procedure.

- **Uex_UIB_Termination**: Contains the entry that is to be called by the compiler at termination time. The user sets this value.
- **Uex_UIB_Message_Filter**: Contains the entry that is to be called by the compiler whenever a message needs to be generated. The user sets this value.

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:

```
Dcl 1 Uex_ISA native based( null() ),
    2 Uex_ISA_Length_fixed bin(31);   /* storage(Uex_IS    A) */
```

The global control block syntax for the initialization procedure is discussed in the section “Structure of global control blocks” on page 232.

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 only decrease the severity of “WARNING” (severity code 4) messages to “INFORMATIONAL” (severity code 0) 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:

```
Dcl 1 Uex_MFA native based( null() ),
      2 Uex_MFA_Length   fixed bin(31),

      2 Uex_MFA_Facility_Id  char(3),        /* of component writing
                                               message             */
      2 *                    char(1),
      2 Uex_MFA_Message_no   fixed bin(31),
      2 Uex_MFA_Severity     fixed bin(15),
      2 Uex_MFA_New_Severity fixed bin(15);  /* set by exit proc    */
```

Data Entry Fields

- **Uex_MFA_Length**: Contains the length of the control block in bytes. The value is storage (Uex_MFA).
- **Uex_MFA_Facility_Id**: Contains the ID of the facility; in this case, the ID is IBM. The compiler sets this value.

- **Uex_MFA_Message_no**: Contains the message number that the compiler is going to generate. The compiler sets this value.
- **Uex_MFA_Severity**: Contains the severity level of the message; it can be from one to fifteen characters in length. The compiler sets this value.
- **Uex_MFA_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.

Upon completion of the message filtering procedure, set the return/reason codes to one of the following:

0/0
Continue compilation, output message

0/1
Continue compilation, do not output message

4/n
Reserved for future use

8/n
Reserved for future use

16/n
Abort compilation

Writing the termination procedure

You should use the termination procedure to perform any cleanup required, such as closing files. You might also want to write out final statistical reports based on information collected during the error message filter procedures and the initialization procedures.

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 232. 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 data conversion tables

The routines that the compiler, preprocessor, library, and debugger use to convert from ASCII to EBCDIC and from EBCDIC to ASCII are found in a separate library file, `libibmrtab.a`.

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. You might want to replace the tables if files are translated from EBCDIC to ASCII as you download them using a table different than the one we ship.

The names of the conversion routines are IBMPBE2A (EBCDIC to ASCII) and IBMPBA2E (ASCII to EBCDIC). Do not change the names of the files shipped with the product.

Chapter 15. Improving performance

Selecting compile-time options for optimal performance 237
OPTIMIZE 237
IMPRECISE 238
GONUMBER 238
RULES 238
PREFIX 239
CONVERSION 239
DEFAULT 239
BYADDR or BYVALUE 240
(NON)CONNECTED 241
RETURNS(BYVALUE) or RETURNS(BYADDR) 241
(NO)DESCRIPTOR 241
(RE)ORDER 241
ASCII or EBCDIC 241
IEEE or HEXADEC 241
(NON)NATIVE 241
(NO)INLINE 242
Summary of compile-time options that improve performance 242
Coding for better performance 242
DATA-directed input and output 243
Input-only parameters 243
String assignments 243
Loop control variables 244
PACKAGEs versus nested PROCEDUREs . . . 244
Example with nested procedures 245
REDUCIBLE functions 245
DEFINED versus UNION 246
Named constants versus static variables . . . 246
Example with optimal code but no meaningful names 246
Avoiding calls to library routines 247

Many considerations for improving the speed of your program are independent of the compiler that you use and the platform on which it runs. This chapter, however, identifies those considerations that are unique to the PL/I for AIX compilers and the code they generate.

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 configuration file.

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.

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:

```
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
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.

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)
NONCONNECTED
DESCRIPTOR
ORDER
ASSIGNABLE
ASCII
IEEE
NATIVE
NODIRECTED
NOINLINE

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::

```
a: proc( parm1, parm2, ..., parmN );

   dcl parm1 byaddr  ...;
   dcl parm2 byvalue ...;
         .
         .
         .
   dcl parmN byvalue ...;

   /* program logic */

end;
```

Faster, equivalent function with BYVALUE parameter::

```
a: proc( parm1, parm2, ..., parmN )
   returns( ... /* attributes of parm1 */ );

   dcl parm1 byvalue  ...;
   dcl parm2 byvalue ...;
         .
         .
         .
   dcl parmN byvalue ...;

   /* program logic */

   return( parm1 );

end;
```

(NON)CONNECTED

The DEFAULT(NONCONNECTED) option indicates that the compiler assumes that any aggregate parameters are NONCONNECTED. References to elements of NONCONNECTED aggregate parameters require the compiler to generate code to access the parameter's descriptor, even if the aggregate is declared with constant extents.

The compiler does not generate these instructions if the aggregate parameter has constant extents and is CONNECTED. Consequently, if your application never passes nonconnected parameters, your code is more optimal if you use the DEFAULT(CONNECTED) option.

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.

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.

Summary of compile-time options that improve performance

In summary, the following options (if appropriate for your application) can improve performance:

OPTIMIZE(2)
IMPRECISE
RULES(ANS NOLAXCTL)
DEFAULT with the following suboptions
(BYVALUE
RETURNS(BYVALUE)
CONNECTED
NODESCRIPTOR
REORDER
ASCII
IEEE
NATIVE

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:

```
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 `getenv` is an input-only CHAR VARYINGZ string:

```
dcl getenv     entry( char(*) varyingz nonasgn byaddr,
                      pointer byaddr )
               returns( native fixed bin(31) optional )
               options( nodescriptor );
```

If this function is invoked with the string 'IBM_OPTIONS',

the compiler can pass the address of that string rather than assigning it to a compiler-generated temporary storage area and passing the address of that area.

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:

```
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
POINTER
OFFSET

Performance also improves if loop control variables are not members of arrays, structures, or unions. The compiler issues a warning message when they are. Loop control variables that are AUTOMATIC and not used for any other purpose give you the optimal code generation.

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

```
a: proc;

  dcl (i,j,k) fixed bin;
  dcl ib      based fixed bin;
      .
      .
      .
  call b( addr(i) );
      .
      .
      .
  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;
end;
```

Example with packaged procedures:

```
p: package exports( a );

  dcl ib      based fixed bin;

  a: proc;

    dcl (i,j,k) fixed bin;
      .
      .
      .
    call b( addr(i) );
      .
      .
      .
  end;

  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;

end p;
```

REDUCIBLE functions

REDUCIBLE indicates that a procedure or 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.

```
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.

```
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 22.

Table 22. Conditions under which conversions are handled inline

Target	Source	Condition
fixed bin(p1,q1)	fixed bin(p2,q2)	always
	float(p2)	if SIZE is disabled
	bit(1)	always
	bit(n) aligned	if n is known and n ≤ 31
	char(1)	
	pic'(n)9'	if CONV is disabled
	pic'(n)Z(m)9'	if n ≤ 6
		if n + m ≤ 6
fixed dec(p1,q1)	fixed dec(p2,q2)	done using an especially fast library routine
float(p1)	fixed bin(p2,q2)	always
	float(p2)	always
	bit(1)	always
	bit(n) aligned	if n is known and n ≤ 31
	char(1)	
	pic'(n)9'	if CONV is disabled
	pic'(n)Z(m)9'	if n ≤ 6
		if n + m ≤ 6
pictured fixed	pictured fixed	if pictures match
pictured float	pictured float	if pictures match
char	char nonvarying	always
	char varying	always
	char varyingz	always
	pictured fixed	always
	pictured float	always
	pictured char	always
pictured char	pictured char	if pictures match
bit(1) nonvarying	bit(1) nonvarying	always
bit(n) nonvarying	bit(m) nonvarying	see note

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 23 on page 249 lists these built-in functions and the conditions under which they are handled inline.

Table 23. Conditions under which string built-in functions are handled inline

String function	Comments and conditions
BOOL	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.
COPY	When the first argument has type character.
EDIT	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.
HIGH	Always
INDEX	When only two arguments are supplied and they have type character.
LENGTH	Always
LOW	Always
MAXLENGTH	Always
SEARCH	When only two arguments are supplied and they have type character.
SEARCHR	When only two arguments are supplied and they have type character.
SUBSTR	When STRINGRANGE is disabled.
TRANSLATE	When the second and third arguments are constant.
TRIM	When only one argument is supplied and it has type character.
UNSPEC	Always
VERIFY	When only two arguments are supplied and they have type character.
VERIFYR	When only two arguments are supplied and they have type character.

Chapter 16. Using PL/I in mixed-language applications

Matching data and linkages 251
What data is passed 252
How data is passed 253
Where data is passed. 255
Maintaining your environment 255
Invoking non-PL/I routines from a PL/I MAIN 255
Invoking PL/I routines from a non-PL/I main . . 256
Using ON ANYCONDITION 256

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 an executable load module written in PL/I. Another possibility is an application using COBOL or FORTRAN, which can load and call PL/I routines packaged in a PL/I executable load module. 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 an executable load modulea 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++, FORTRAN, COBOL, and PL/I) (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

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 24 lists the scalar data types which are equivalent between PL/I and C.

Table 24. Equivalent data types between C and PL/I

C Data Type	PL/I Data Type
signed char	FIXED BIN(7,0)
unsigned char	UNSIGNED FIXED BIN(8,0) or CHAR(1)
signed short	FIXED BIN(15,0)
unsigned short	UNSIGNED FIXED BIN(16,0)
signed (long) int	FIXED BIN(31,0)
unsigned (long) int	UNSIGNED FIXED BIN(31,0)
float	FLOAT BIN(21) FLOAT DEC(6)
double	FLOAT BIN(53) FLOAT DEC(16)
long double	FLOAT BIN(64) FLOAT DEC 18)
enum	ORDINAL
<non-function-type> *	POINTER or HANDLE
<function-type> *	ENTRY LIMITED

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:

```
short x [ 6 ];
```

In PL/I, the array would be declared as follows:

```
dcl 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, while 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

```
struct { unsigned byte1 :8;
         unsigned byte2 :8;
         unsigned byte3 :8;
         unsigned byte4 :8;
       } bytes;
```

PL/I Structure

```
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 X'00'. Thus, in the example below, *address* is a C 'string' that could hold up to 30 non-null characters.

```
char address [ 31 ];
```

The following PL/I declare most closely resembles the C 'string'.

```
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:

```
int strcspn( char * string1, char * string2 );
```

The PL/I declare for the same function would be:

```
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.

```
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.

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.

```
dcl modf entry( float bin(53) byvalue,
                float bin(53) byaddr )
          returns( float bin(53) );
```

The corresponding C declaration is:

```
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.

```
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:

```
dcl strcspn entry( nonasgn char(*) varyingz,
                   nonasgn char(*) varyingz )
             returns( fixed bin(31) );
```

The corresponding C declaration is:

```
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.

```
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 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 strcspn 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.

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 coded in PL/I, you can call two kinds of non-PL/I routines:

- System routines (such as DOS and Windows services)
- C, COBOL, or REXX routines

System routines do not require their own run-time 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 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

```
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

```
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 executable module is dynamically loaded from a non-PL/I main program.
- Exist with a non-PL/I language run-time environment with minimal conflicts.

Any PL/I routine called directly from non-PL/I routines must have the FROMALIEN option in the OPTIONS option and must not specify the MAIN option.

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 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:

```
pliapp:
  proc( p1, ...,  pn )
  returns( ... )
  options( fromalien );

/* declarations of paramaters, if any */

/* declarations of other variables    */

on anycondition
  begin;
    /* handle condition if possible   */

    /* if unhandled, set return value */
    goto return_stmt;
  end;
```

```
/* mainline code */

return_Stmt:
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 17. Using sort routines

Comparing S/390 and workstation sort programs 259
Preparing to use sort 260
Choosing the type of sort 261
Specifying the sorting field 263
Example: 264
Specifying the records to be sorted 264
Example: 264
Calling the sort program 264
PLISRT examples 264
Example 1 264
Example 2 265
Example 3 265
Example 4 265
Determining whether the sort was successful 265
Sort data input and output 266
Sort data handling routines. 266
E15 — input-handling routine (sort exit E15) 266
E35 — output-handling routine (sort exit E35) 269
Calling PLISRTA 271
Calling PLISRTB 272
Calling PLISRTC 274
Calling PLISRTD, example 1 275
Calling PLISRTD, example 2 276

PL/I for AIX PL/I for Windows supports the PLISRT*x* (*x* = A, B, C, or D) built-in subroutines. To use the PLISRT*x* 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 25. workstation PLISRTx

Built-in subroutine	Arguments
PLISRTA Sort input: data set Sort output: data set	(sort statement,record statement,storage,return code [,data set prefix,message level, sort technique])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(sort statement,record statement,storage,return code, input routine [,data set prefix,message level,sort technique])
PLISRTC Sort input: data set Sort output: PL/I subroutine	(sort statement,record statement,storage,return code, output routine [,data set prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code, input routine,output routine [,data set prefix,message level,sort technique])

Table 25. workstation PLISRTx (continued)

Built-in subroutine	Arguments

Argument definitions:

Sort statement
Character string expression describing sorting fields and format. See “Specifying the sorting field” on page 263.

Record statement
Character string expression describing the length and record format of data. See “Specifying the records to be sorted” on page 264.

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 266.

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 269.

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 266.

The sort built-in subroutines and the source and destination of data are as follows:

Built-in subroutine	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine

Built-in subroutine	Source	Destination
PLISRTD	Subroutine	Subroutine

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 PLISRT*x*.

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 PLISRT*x* statement must contain an argument list giving the sort program information about the data set to be sorted, the fields on which it is to be sorted, the 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 of 0 or 16 to the PL/I calling procedure.
2. Raising an error condition when certain errors are detected and the return code is undefined.

Figure 26 on page 262 is a simplified flowchart showing the sort operation.

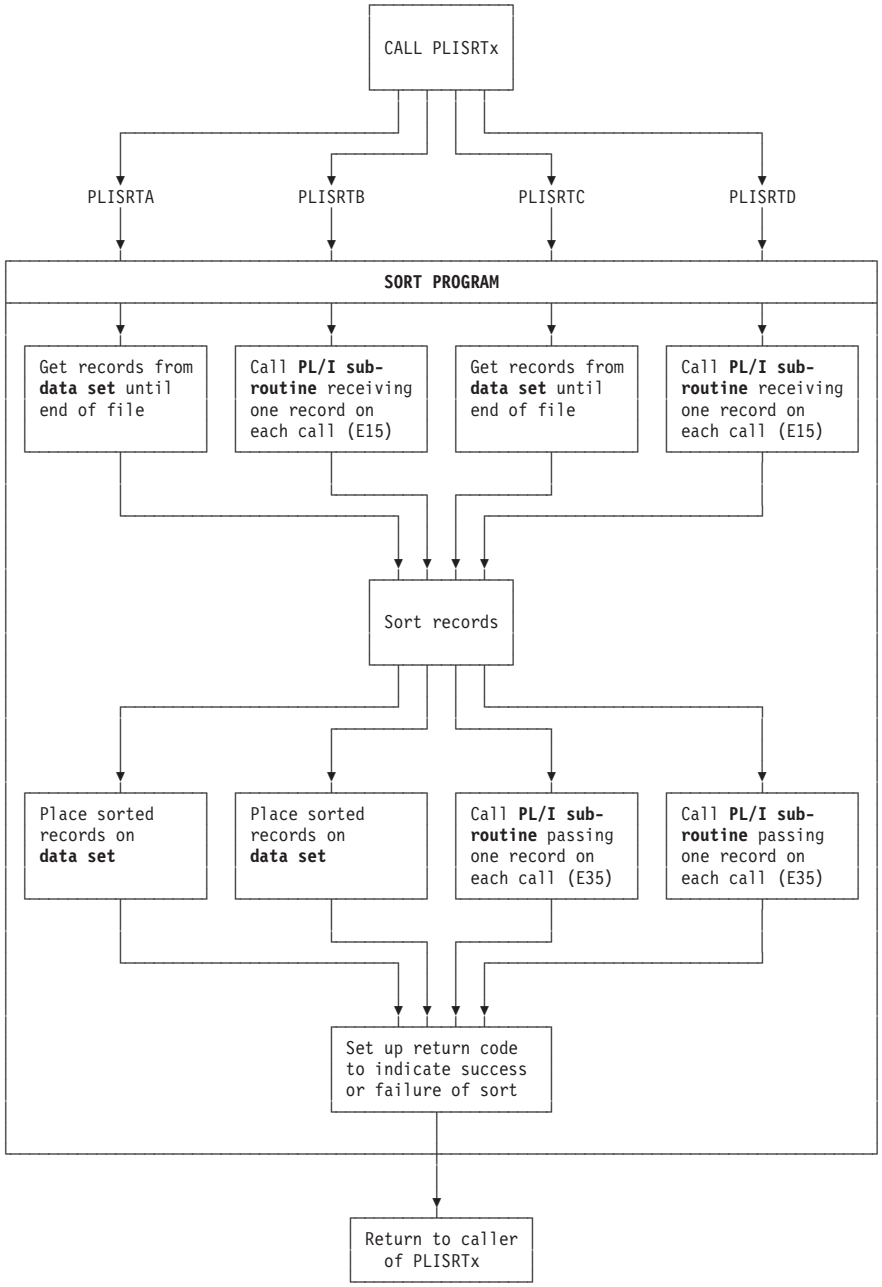


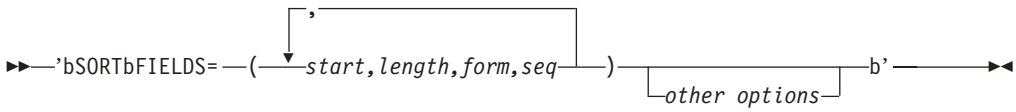
Figure 26. Flow of control for the sort program

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.

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:



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 arbitrary. Fields can overlay each other.

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.

Code	Data Type and Length
CH	character 1–256
ZD	zoned decimal signed 1–32
PD	packed decimal signed 1–32
FI	fixed point, signed 1–256
BI	binary, unsigned 1 bit to 256 bytes

The sum of the lengths of all fields must not exceed 256 bytes.

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.

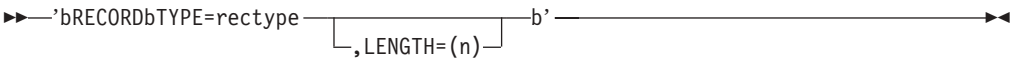
other options
The only option supported under workstation PL/I is the default, EQUALS. Source code downloaded from the mainframe, however, does not need to be altered.

Example:

```
' SORT FIELDS=(1,10,CH,A) '
```

Specifying the records to be sorted

Use the RECORD statement as the second argument to PLISRT*x*. The syntax of the RECORD statement must be a character string expression which, when evaluated, accepts the following syntax:



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 PLISRT*x* 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).

```
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 FIELD =(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 plisrtd (' 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;
```

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:

```
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.

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 30 on page 271. 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 datasets 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 dataset 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 PLISRT*x*, 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 PLISRT*x*.

The routines are called individually for each record required by sort or passed from sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures 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.

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 31 on page 272 and Figure 33 on page 275), 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 31 on page 272 and Figure 33 on page 275.

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 PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 32 on page 274.

```
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      */

       /*    The code to do your processing goes here    */

       call pliretc (12);/* Set return code of 12, meaning this
                            record is to be included in the sort        */
       return (string);  /* Return data with RETURN statement           */
   end;
  end;                   /* End of the input procedure                  */
```

Figure 27. Skeletal code for an input procedure

In addition, to code the input user exit routine, the explicit attributes of the E15 must be specified in the program unit that calls PLISRTx if E15 is not nested in that program unit.

```
  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 28. 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 32 on page 274

274 and Figure 33 on page 275, or to use the sorted data to generate further information. The output handling routine is used by sort when a call is made to PLISRTC or PLISRTD.

When the records have been sorted, sort passes them (one at a time) to the output handling routine. The output routine then processes them as required. When all the records have been passed, sort sets up its return code and returns to the statement after the CALL PLISRT*x* statement. 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 PLISRT*x*.

The record is passed from sort to the output routine as a character string, and you must declare a character string parameter in the output-handling subroutine to receive the data. The output-handling subroutine must also pass 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 PLISRT*x*, 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 29.

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 PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples at the end of this chapter.

```
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 29. 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:                                  */
/*    export DD_SORTIN='ex106.dat,type(crlf),lrecl(80),delimit(y)' */
/*    export DD_SORTOUT='ex106.out,type(crlf),lrecl(80),delimit(y)'*/
/*                                                                 */
/*                                                                 */
/*******************************************************************/

  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 30. PLISRTA—Sorting from input data set to output data set

Content of EX106.DAT to be used with Figure 30

```
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 PLISRTB

```
/*******************************************************************/
/*                                                                 */
/*  DESCRIPTION                                                    */
/*    Sorting from an input-handling routine to an output data set */
/*                                                                 */
/*  Use the following statements:                                  */
/*                                                                 */
/*    export DD_SYSIN='ex107.dat,type(crlf),lrecl(80),delimit(y)'  */
/*    export DD_SORTOUT='ex107.out,type(crlf),lrecl(80),delimit(y)'*/
/*                                                                 */
/*******************************************************************/
 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 31. PLISRTB—Sorting from input-handling routine to output data set

Content of EX107.DAT to be used with Figure 31 on page 272

```
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 PLISRTC

```
/*******************************************************************/
/*                                                                 */
/*  DESCRIPTION                                                    */
/*    Sorting from an input data set to an output-handling routine */
/*                                                                 */
/*  Use the following statement:                                   */
/*                                                                 */
/*    export DD_SORTIN='ex108.dat,type(crlf),lrecl(80),delimit(y)' */
/*                                                                 */
/*******************************************************************/

 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 */;
      /* Set pl/i return code to reflect success of sort       */
     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 32. PLISRTC—Sorting from input data set to output-handling routine

Content of EX108.DAT to be used with Figure 32

```
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 1

```
/*******************************************************************/
/*                                                                 */
/*  DESCRIPTION                                                    */
/*    Sorting an input-handling to output-handling routine         */
/*                                                                 */
/*  Use the following statement:                                   */
/*                                                                 */
/*    export 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 33. PLISRTD—Sorting input-handling routine to output-handling routine

Contents of EX109.DAT and EX110.DAT used with Figure 33 and Figure 34 on page 276

```
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

```
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 34. PLISRTD—Sorting input-handling routine to output-handling routine

Chapter 18. 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.

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.

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 USS, the character string specifying the input file should have the form 'file:///filename', where filename is the name of a USS 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 35 on page 279. In these descriptions, the term ″XML text″ refers to the string based on the pointer and length passed to the event.

```
xmlDocument =
   '<?xml version="1.0" standalone="yes"?>'
|| '<!--This document is just an example-->'
|| '<sandwich>'
|| '<bread type="baker'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';
```

Figure 35. Sample XML document

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 ″&#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.

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.

content_predefined_reference

This event occurs in element content for the five pre-defined entity references ″&″, ″’″, ″>″, ″<″ and ″’″. The parser passes a CHAR(1) or WIDECHAR(1) value that contains one of ″&″, ″’″, ″>″, ″<″ or ’″’, respectively.

content_character_reference

This event occurs in element content for numeric character references (Unicode code points or ″scalar values″) of the form ″&#dd;″ or ″&#xhh;″, where ″d″ 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 WIDECHAR 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).

CCSID	Description
01047	Latin 1 / Open Systems
01140, 00037	USA, Canada, etc.
01141, 00273	Austria, Germany
01142, 00277	Denmark, Norway
01143, 00278	Finland, Sweden
01144, 00280	Italy
01145, 00284	Spain, Latin America (Spanish)
01146, 00285	UK
01147, 00297	France
01148, 00500	International

CCSID	Description
01149, 00871	Iceland

Supported ASCII code pages

CCSID	Description
00813	ISO 8859-7 Greek / Latin
00819	ISO 8859-1 Latin 1 / Open Systems
00920	ISO 8859-9 Latin 5 (ECMA-128, Turkey TS-5881)

Specifying the code page

If your document does not include an encoding declaration in the XML declaration, or 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 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):

IBM_	CP	CCSID_
IBM-	CP_	CCSID-
	CP-	

Using an alias

You can use any of the following supported aliases (in any mixture of lower and upper case):

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
813	ISO-8859-7, ISO_8859-7
819	ISO-8859-1, ISO_8859-1
920	ISO-8859-9, ISO_8859-9

Code page	Supported aliases
1200	UTF-16

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 nodescriptor );

  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 36. PLISAXA coding example - type declarations

```
saxtest: proc options( main );

  dcl
    1 eventHandler static

      ,2 e01 type event
             init( start_of_document )
      ,2 e02 type event
             init( version_information )
      ,2 e03 type event
             init( encoding_declaration )
      ,2 e04 type event
             init( standalone_declaration )
      ,2 e05 type event
             init( document_type_declaration )
      ,2 e06 type event_end_of_document
             init( end_of_document )
      ,2 e07 type event
             init( start_of_element )
      ,2 e08 type event
             init( attribute_name )
      ,2 e09 type event
             init( attribute_characters )
      ,2 e10 type event_predefined_ref
             init( attribute_predefined_reference )
      ,2 e11 type event_character_ref
             init( attribute_character_reference )
      ,2 e12 type event
             init( end_of_element )
      ,2 e13 type event
             init( start_of_CDATA )
      ,2 e14 type event
             init( end_of_CDATA )
      ,2 e15 type event
             init( content_characters )
      ,2 e16 type event_predefined_ref
             init( content_predefined_reference )
      ,2 e17 type event_character_ref
             init( content_character_reference )
      ,2 e18 type event_pi
             init( processing_instruction )
      ,2 e19 type event
             init( comment )
      ,2 e20 type event
             init( unknown_attribute_reference )
      ,2 e21 type event
             init( unknown_content_reference )
      ,2 e22 type event
             init( start_of_prefix_mapping )
      ,2 e23 type event
             init( end_of_prefix_mapping )
      ,2 e24 type event_exception
             init( exception )
      ;
```

Figure 37. 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 38. 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 39. PLISAXA coding example - event routines (Part 1 of 8)

```
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;

end_of_document:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;

  put skip list( lowercase( procname() ) );

  return(0);
end;
```

Figure 39. PLISAXA coding example - event routines (Part 2 of 8)

```
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,1,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,1,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,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 39. PLISAXA coding example - event routines (Part 3 of 8)

```
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;
```

Figure 39. PLISAXA coding example - event routines (Part 4 of 8)

```
start_of_CDATA:
  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_of_CDATA:
  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;
```

Figure 39. PLISAXA coding example - event routines (Part 5 of 8)

```
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;
```

Figure 39. PLISAXA coding example - event routines (Part 6 of 8)

```
comment:
  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;

unknown_attribute_reference:
  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;

unknown_content_reference:
  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 39. PLISAXA coding example - event routines (Part 7 of 8)

```
start_of_prefix_mapping:
  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_of_prefix_mapping:
  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;

exception:
  proc( userToken, xmlToken, currentOffset, errorID )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl currentOffset  fixed bin(31);
  dcl errorID        fixed bin(31);

  put skip list( lowercase( procname() )
   || ' errorid =' || errorid );

  return(0);
end;
end;
```

Figure 39. PLISAXA coding example - event routines (Part 8 of 8)

The preceding program would produce the following output:

```
start_of_dcoument length=           305
version_information <1.0>
standalone_declaration <yes>
comment <This document is just an example>
start_of_element <sandwich>
start_of_element <bread>
attribute_name <type>
attribute_characters <baker>
attribute_predefined_reference 7D
attribute_characters <s best>
end_of_element <bread>
processing_instruction <spread>
start_of_element <meat>
content_characters <Ham >
content_predefined_reference 50
content_characters < turkey>
end_of_element <meat>
start_of_element <filling>
content_characters <Cheese, lettuce, tomato, etc.>
end_of_element <filling>
start_of_cdata <<![CDATA[>
content_characters <We should add a <relish> element in future!>
end_of_cdata <]]>
end_of_element <sandwich>
exception errorid =             1
content_characters <j>
exception errorid =             1
content_characters <u>
exception errorid =             1
content_characters <n>
exception errorid =             1
content_characters <k>
end_of_document
```

Figure 40. PLISAXA coding example - program 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 26. Continuable Exceptions

Number	Description	Parser Action on Continuation
1	The parser found an invalid character while scanning white space outside element content.	The parser generates a content_characters event with XML text containing the (single) invalid character. Parsing continues at the character after the invalid character.
2	The parser found an invalid start of a processing instruction, element, comment or document type declaration outside element content.	The parser generates a content_characters event with the XML text containing the 2- or 3-character invalid initial character sequence. Parsing continues at the character after the invalid sequence.

Table 26. Continuable Exceptions (continued)

Number	Description	Parser Action on Continuation
3	The parser found a duplicate attribute name.	The parser generates an attribute_name event with the XML text containing the duplicate attribute name.
4	The parser found the markup character ″<″ in an attribute value.	Prior to generating the exception event, the parser generates an attribute_characters event for any part of the attribute value prior to the ″<″ character. After the exception event, the parser generates an attribute_characters event with XML text containing ″<″. Parsing then continues at the character after the ″<″.
5	The start and end tag names of an element did not match.	The parser generates an end_of_element event with XML text containing the mismatched end name.
6	The parser found an invalid character in element content.	The parser includes the invalid character in XML text for the subsequent content_characters event.
7	The parser found an invalid start of an element, comment, processing instruction or CDATA section in element content.	Prior to generating the exception event, the parser generates a content_characters event for any part of the content prior to the ″<″ markup character. After the exception event, the parser generates a content_characters event with XML text containing 2 characters: the ″<″ followed by the invalid character. Parsing continues at the character after the invalid character.
8	The parser found in element content the CDATA closing character sequence “]]” without the matching opening character sequence “<![CDATA[”.	Prior to generating the exception event, the parser generates a content_characters event for any part of the content prior to the “]]” character sequence. After the exception event, the parser generates a content_characters event with XML text containing the 3-character sequence “]]”. Parsing continues at the character after this sequence.
9	The parser found an invalid character in a comment.	The parser includes the invalid character in XML text for the subsequent comment event.
10	The parser found in a comment the character sequence ″--″ not followed by ″>″.	The parser assumes that the ″--″ character sequence terminates the comment, and generates a comment event. Parsing continues at the character after the ″--″ sequence.
11	The parser found an invalid character in a processing instruction data segment.	The parser includes the invalid character in XML text for the subsequent processing_instruction event.

Table 26. Continuable Exceptions (continued)

Number	Description	Parser Action on Continuation
12	A processing instruction target name was ″xml″ in lower-case, upper-case or mixed-case.	The parser generates a processing_instruction event with XML text containing ″xml″ in the original case.
13	The parser found an invalid digit in a hexadecimal character reference (of the form &#xdddd;).	The parser generates an attribute_characters or content_characters event with XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.
14	The parser found an invalid digit in a decimal character reference (of the form &#dddd;).	The parser generates an attribute_characters or content_characters event with XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.
15	The encoding declaration value in the XML declaration did not begin with lower- or upper-case A through Z	The parser generates the encoding event with XML text containing the encoding declaration value as it was specified.
16	A character reference did not refer to a legal XML character.	The parser generates an attribute_character_reference or content_character_reference event with XML-NTEXT containing the single Unicode character specified by the character reference.
17	The parser found an invalid character in an entity reference name.	The parser includes the invalid character in the XML text for the subsequent unknown_attribute_reference or unknown_content_reference event.
18	The parser found an invalid character in an attribute value.	The parser includes the invalid character in XML text for the subsequent attribute_characters event.
50	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
51	The document was encoded in EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
52	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an ASCII encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.

Table 26. Continuable Exceptions (continued)

Number	Description	Parser Action on Continuation
53	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
54	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
55	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
56	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
57	The document was encoded in ASCII, and the document encoding declaration specified a supported ASCII encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
58	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported EBCDIC encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
59	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
60	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.

Table 26. Continuable Exceptions (continued)

Number	Description	Parser Action on Continuation
61	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
100,001 through 165,535	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.	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.
200,001 through 265,535	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.	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.

Terminating exception codes

Table 27. Terminating Exceptions

Number	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.

Table 27. Terminating Exceptions (continued)

Number	Description
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, '_' or ':'.
125	The first character of the first attribute name of an element was not a letter, '_' or ':'.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than '=' following an attribute name.
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, '_' or ':'.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a '>' following the '/'.
133	The first character of an element end tag name was not a letter, '_' or ':'.
134	An element end tag name was not terminated by a '>'.
135	The first character of an element name was not a letter, '_' or ':'.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, '_' or ':'.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence '?>'.
141	The parser found an invalid character following '&' in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by a '='.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.

Table 27. Terminating Exceptions (continued)

Number	Description
148	'encoding' in the XML declaration was not followed by a '='.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional standalone declaration in the XML declaration.
153	'standalone' in the XML declaration was not followed by a '='.
154	The standalone declaration value in the XML declaration is either missing or improperly delimited.
155	The standalone declaration value was neither 'yes' nor 'no' only.
156	The standalone declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
157	The parser found an invalid character following the standalone declaration value closing delimiter in the XML declaration.
158	The XML declaration was not terminated by the proper character sequence '?>', or contained an invalid attribute.
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
300	The document was encoded in EBCDIC, but the CODEPAGE compiler option specified a supported ASCII code page.
301	The document was encoded in EBCDIC, but the CODEPAGE compiler option specified Unicode.
302	The document was encoded in EBCDIC, but the CODEPAGE compiler option specified an unsupported code page.
303	The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document encoding declaration was either empty or contained an unsupported alphabetic encoding alias.
304	The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document did not contain an encoding declaration.
305	The document was encoded in EBCDIC, but the CODEPAGE compiler option is unsupported and the document encoding declaration did not specify a supported EBCDIC encoding.
306	The document was encoded in ASCII, but the CODEPAGE compiler option specified a supported EBCDIC code page.
307	The document was encoded in ASCII, but the CODEPAGE compiler option specified Unicode.
308	The document was encoded in ASCII, but the CODEPAGE compiler option did not specify a supported EBCDIC code page, ASCII or Unicode.
309	The CODEPAGE compiler option specified a supported ASCII code page, but the document was encoded in Unicode.
310	The CODEPAGE compiler option specified a supported EBCDIC code page, but the document was encoded in Unicode.
311	The CODEPAGE compiler option specified an unsupported code page, but the document was encoded in Unicode.

Table 27. Terminating Exceptions (continued)

Number	Description
312	The document was encoded in ASCII, but both the encodings provided externally and within the document encoding declaration are unsupported.
313	The document was encoded in ASCII, but the CODEPAGE compiler option is unsupported and the document did not contain an encoding declaration.
314	The document was encoded in ASCII, but the CODEPAGE compiler option is unsupported and the document encoding declaration did not specify a supported ASCII encoding.
315	The document was encoded in UTF-16 Little Endian, which the parser does not support on this platform.
316	The document was encoded in UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document may be damaged.
318	The document was encoded in UTF-8, which the parser does not support.
319	The document was encoded in UTF-16 Big Endian, which the parser does not support on this platform.
500 to 99,999	Internal error. Please report the error to your service representative.

Part 6. Appendixes

Product specifications and service

Product specifications and service

This appendix provides information on the operating environment and installation for PL/I for AIX as well as information on what to do if you have questions or comments about the product.

Specified operating environment

The following hardware and software requirements apply to PL/I for AIX.

Hardware requirements

PL/I for AIX Version 1 and its generated object programs run programs run on RISC System/6000 family processors running IBM AIX Version 4.1.3 (or subsequent releases), including POWER, POWER2, and PowerPC processors, configured with at least one supported display, keyboard, and mouse.

Software requirements

Required programs: PL/I for AIX Version 1 and its generated object programs run under the following system environment (or subsequent releases):

- IBM AIX Version 4.1.3.

Optional programs: In order to use PL/I for AIX with optional programs such as DB2 and CICS, additional products are required.

Installing PL/I for AIX

PL/I for AIX is available on an 8mm tape or on CD-ROM. For detailed information about installation, see the Getting Started document provided with the product.

Shipping shared runtime libraries

With PL/I for AIX, you can ship the shared runtime libraries with your applications at no cost to you. If you are reshipping a PL/I application, you need to include the following files with your application:

libplishr.a
pli_ddm
plirun.cat

There is also a runtime library (`libibmrtab.a`) that is necessary to run any program. If you are shipping `libplishr.a` with an application that you create, you must also ship `libibmrtab.a`.

If you are reshipping a PL/I thread safe application, you need to include the following files with your application:

libplishr_r.a
pli_ddm
plirun.cat

If you are using VSAM, you should also include:

libdubaix.a

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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
555 Bailey Avenue
San Jose, CA 95141-1099

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations might not appear.

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 MVS & VM.

Macros for customer use

IBM PL/I for MVS & VM provides no macros that allow a customer installation to write programs that use the services of IBM PL/I for MVS & VM.

Attention: Do not use as programming interfaces any IBM PL/I for MVS & VM macros.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	IMS/ESA
CICS	Language Environment
CICS/ESA	OS/2
DFSMS/MVS	OS/390
DFSORT	Proprinter
IBM	VisualAge
IMS	WebSphere

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

Enterprise PL/I publications

Programming Guide, SC27-1457
Language Reference, SC27-1460
Messages and Codes, SC27-1461
Diagnosis Guide, GC27-1459
Compiler and Run-Time Migration Guide, GC27-1458

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

CICS Transaction Server

Customization Guide, SC33-1683
External Interfaces Guide, SC33-1944
Application Programming Reference, SC33-1688
Application Programming Guide, SC33-1687

Glossary

This glossary defines terms for all platforms and releases of PL/I. It might contain terms that this manual does not use. If you do not find the terms for which you are looking, see the index in this manual or *IBM Dictionary of Computing*, SC20-1699.

A

access. To reference or retrieve data.

action specification. In an ON statement, the ON-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever the appropriate condition is raised.

activate (a block). To initiate the execution of a block. A procedure block is activated when it is invoked. A begin-block is activated when it is encountered in the normal flow of control, including a branch. A package cannot be activated.

activate (a preprocessor variable or preprocessor entry point). To make a macro facility identifier eligible for replacement in subsequent source code. The %ACTIVATE statement activates preprocessor variables or preprocessor entry points.

active. The state of a block after activation and before termination. The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text. The state in which an event variable is said to be during the time it is associated with an asynchronous operation. The state in which a task variable is said to be when its associated task is attached. The state in which a task is said to be before it has been terminated.

actual origin (AO). The location of the first item in the array or structure.

additive attribute. A file description attribute for which there are no defaults, and which, if required, must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute.*

adjustable extent. The bound (of an array), the length (of a string), or the size (of an area) that might be different for different generations of the associated variable. Adjustable extents are specified as expressions or asterisks (or by REFER options for based variables), which are evaluated separately for each generation. They cannot be used for static variables.

aggregate. See *data aggregate.*

aggregate expression. An array, structure, or union expression.

aggregate type. For any item of data, the specification whether it is structure, union, or array.

allocated variable. A variable with which main storage is associated and not freed.

allocation. The reservation of main storage for a variable. A generation of an allocated variable. The association of a PL/I file with a system data set, device, or file.

alignment. The storing of data items in relation to certain machine-dependent boundaries (for example, a fullword or halfword boundary).

alphabetic character. Any of the characters A through Z of the English alphabet and the alphabetic extenders #, $, and @ (which can have a different graphic representation in different countries).

alphameric character. An alphabetic character or a digit.

alternative attribute. A file description attribute that is chosen from a group of attributes. If none is specified, a default is assumed. Contrast with *additive attribute.*

ambiguous reference. A reference that is not sufficiently qualified to identify one and only one name known at the point of reference.

area. A portion of storage within which based variables can be allocated.

argument. An expression in an argument list as part of an invocation of a subroutine or function.

argument list. A parenthesized list of zero or more arguments, separated by commas, following an entry name constant, an entry name variable, a generic name, or a built-in function name. The list becomes the parameter list of the entry point.

arithmetic comparison. A comparison of numeric values. See also *bit comparison, character comparison*.

arithmetic constant. A fixed-point constant or a floating-point constant. Although most arithmetic constants can be signed, the sign is not part of the constant.

arithmetic conversion. The transformation of a value from one arithmetic representation to another.

arithmetic data. Data that has the characteristics of base, scale, mode, and precision. Coded arithmetic data and pictured numeric character data are included.

arithmetic operators. Either of the prefix operators + and −, or any of the following infix operators: + − * / **

array. A named, ordered collection of one or more data elements with identical attributes, grouped into one or more dimensions.

array expression. An expression whose evaluation yields an array of values.

array of structures. An ordered collection of identical structures specified by giving the dimension attribute to a structure name.

array variable. A variable that represents an aggregate of data items that must have identical attributes. Contrast with *structure variable.*

ASCII. American National Standard Code for Information Interchange.

assignment. The process of giving a value to a variable.

asynchronous operation. The overlap of an input/output operation with the execution of statements. The concurrent execution of procedures using multiple flows of control for different tasks.

attachment of a task. The invocation of a procedure and the establishment of a separate flow of control to execute the invoked procedure (and procedures it invokes) asynchronously, with execution of the invoking procedure.

attention. An occurrence, external to a task, that could cause a task to be interrupted.

attribute. A descriptive property associated with a name to describe a characteristic represented. A descriptive property used to describe a characteristic of the result of evaluation of an expression.

automatic storage allocation. The allocation of storage for automatic variables.

automatic variable. A variable whose storage is allocated automatically at the activation of a block and released automatically at the termination of that block.

B

base. The number system in which an arithmetic value is represented.

base element. A member of a structure or a union that is itself not another structure or union.

base item. The automatic, controlled, or static variable or the parameter upon which a defined variable is defined.

based reference. A reference that has the based storage class.

based storage allocation. The allocation of storage for based variables.

based variable. A variable whose storage address is provided by a locator. Multiple generations of the same variable are accessible. It does not identify a fixed location in storage.

begin-block. A collection of statements delimited by BEGIN and END statements, forming a name scope. A begin-block is activated either by the raising of a condition (if the begin-block is the action specification for an ON-unit) or through the normal flow of control, including any branch resulting from a GOTO statement.

binary. A number system whose only numerals are 0 and 1.

binary digit. See *bit*.

binary fixed-point value. An integer consisting of binary digits and having an optional binary point and optional sign. Contrast with *decimal fixed-point value.*

binary floating-point value. An approximation of a real number in the form of a significand, which can be considered as a binary fraction, and an exponent, which can be considered as an integer exponent to the base of 2. Contrast with *decimal floating-point value.*

bit. A 0 or a 1. The smallest amount of space of computer storage.

bit comparison. A left-to-right, bit-by-bit comparison of binary digits. See also *arithmetic comparison*, *character comparison*.

bit string constant. A series of binary digits enclosed in and followed immediately by the suffix B. Contrast with *character constant*. A series of hexadecimal digits enclosed in single quotes and followed by the suffix B4.

bit string. A string composed of zero or more bits.

bit string operators. The logical operators not and exclusive-or (¬), and (&), and or (|).

bit value. A value that represents a bit type.

block. A sequence of statements, processed as a unit, that specifies the scope of names and the allocation of storage for names declared within it. A block can be a package, procedure, or a begin-block.

bounds. The upper and lower limits of an array dimension.

break character. The underscore symbol (_). It can be used to improve the readability of identifiers. For instance, a variable could be called OLD_INVENTORY_TOTAL instead of OLDINVENTORYTOTAL.

built-in function. A predefined function supplied by the language, such as SQRT (square root).

built-in function reference. A built-in function name, which has an optional argument list.

built-in name. The entry name of a built-in subroutine.

built-in subroutine. Subroutine that has an entry name that is defined at compile-time and is invoked by a CALL statement.

buffer. Intermediate storage, used in input/output operations, into which a record is read during input and from which a record is written during output.

C

call. To invoke a subroutine by using the CALL statement or CALL option.

character comparison. A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison, bit comparison*.

character string constant. A sequence of characters enclosed in single quotes; for example, 'Shakespeare''s 'Hamlet:''.

character set. A defined collection of characters. See *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 listng of the job control statements and job scheduler messages and contains return code indicating success or failure for each step.

entry constant. The label prefix of a PROCEDURE statement (an entry name). The declaration of a name with the ENTRY attribute but without the VARIABLE attribute.

entry data. A data item that represents an entry point to a procedure.

entry expression. An expression whose evaluation yields an entry name.

entry name. An identifier that is explicitly or contextually declared to have the ENTRY attribute (unless the VARIABLE attribute is given) or An identifier that has the value of an entry variable with the ENTRY attribute implied.

entry point. A point in a procedure at which it can be invoked. *primary entry point* and *secondary entry point*.

entry reference. An entry constant, an entry variable reference, or a function reference that returns an entry value.

entry variable. A variable to which an entry value can be assigned. It must have both the ENTRY and VARIABLE attributes.

entry value. The entry point represented by an entry constant or variable; the value includes the environment of the activation that is associated with the entry constant.

environment (of an activation). Information associated with and used in the invoked block regarding data declared in containing blocks.

environment (of a label constant). Identity of the particular activation of a block to which a reference to a statement-label constant applies. This information is determined at the time a statement-label constant is passed as an argument or is assigned to a statement-label variable, and it is passed or assigned along with the constant.

established action. The action taken when a condition is raised. See also *implicit action* and *ON-statement action*.

epilogue. Those processes that occur automatically at the termination of a block or task.

evaluation. The reduction of an expression to a single value, an array of values, or a structured set of values.

event. An activity in a program whose status and completion can be determined from an associated event variable.

event variable. A variable with the EVENT attribute that can be associated with an event. Its value indicates whether the action has been completed and the status of the completion.

explicit declaration. The appearance of an identifier (a name) in a DECLARE statement, as a label prefix, or in a parameter list. Contrast with *implicit declaration*.

exponent characters. The following picture specification characters:

1. K and E, which are used in floating-point picture specifications to indicate the beginning of the exponent field.
2. F, the scaling factor character, specified with an integer constant that indicates the number of decimal positions the decimal point is to be moved from its assumed position to the right (if the constant is positive) or to the left (if the constant is negative).

expression. A notation, within a program, that represents a value, an array of values, or a structured set of values. A constant or a reference appearing alone, or a combination of constants and/or references with operators.

extended alphabet. The uppercase and lowercase alphabetic characters A through Z, $, @ and #, or those specified in the NAMES compiler option.

extent. The range indicated by the bounds of an array dimension, by the length of a string, or by the size of an area. The size of the target area if this area were to be assigned to a target area.

external name. A name (with the EXTERNAL attribute) whose scope is not necessarily confined only to one block and its contained blocks.

external procedure. A procedure that is not contained in any other procedure. A level-2 procedure contained in a package that is also exported.

external symbol. Name that can be referred to in a control section other than the one in which it is defined.

External Symbol Dictionary (ESD). Table containing all the external symbols that appear in the object module.

extralingual character. Characters (such as $, @, and #) that are not classified as alphanumeric or special. This group includes characters that are determined with the NAMES compiler option.

F

factoring. The application of one or more attributes to a parenthesized list of names in a DECLARE statement, eliminating the repetition of identical attributes for multiple names.

field (in the data stream). That portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

field (of a picture specification). Any character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number.

file. A named representation, within a program, of a data set or data sets. A file is associated with the data set(s) for each opening.

file constant. A name declared with the FILE attribute but not the VARIABLE attribute.

file description attributes. Keywords that describe the individual characteristics of each file constant. See also *alternative attribute* and *additive attribute*.

file expression. An expression whose evaluation yields a value of the type file.

file name. A name declared for a file.

file variable. A variable to which file constants can be assigned. It has the attributes FILE and VARIABLE and cannot have any of the file description attributes.

fixed-point constant. See *arithmetic constant*.

fix-up. A solution, performed by the compiler after detecting an error during compilation, that allows the compiled program to run.

floating-point constant. See *arithmetic constant*.

flow of control. Sequence of execution.

format. A specification used in edit-directed data transmission to describe the representation of a data item in the stream (data format item) or the specific positioning of a data item within the stream (control format item).

format constant. The label prefix on a FORMAT statement.

format data. A variable with the FORMAT attribute.

format label. The label prefix on a FORMAT statement.

format list. In stream-oriented transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

fully qualified name. A name that includes all the names in the hierarchical sequence above the member to which the name refers, as well as the name of the member itself.

function (procedure). A procedure that has a RETURNS option in the PROCEDURE statement. A name declared with the RETURNS attribute. It is invoked by the appearance of one of its entry names in a function reference and it returns a scalar value to the point of reference. Contrast with *subroutine*.

function reference. An entry constant or an entry variable, either of which must represent a function, followed by a possibly empty argument list. Contrast with *subroutine call*.

G

generation (of a variable). The allocation of a static variable, a particular allocation of a controlled or automatic variable, or the storage indicated by a particular locator qualification of a based variable or by a defined variable or parameter.

generic descriptor. A descriptor used in a GENERIC attribute.

generic key. A character string that identifies a class of keys. All keys that begin with the string are members of that class. For example, the recorded keys 'ABCD', 'ABCE', and 'ABDF', are all members of the classes identified by the generic keys 'A' and 'AB', and the first two are also members of the class 'ABC'; and the three recorded keys can be considered to be unique members of the classes 'ABCD', 'ABCE', 'ABDF', respectively.

generic name. The name of a family of entry names. A reference to the generic name is replaced by the entry name whose parameter descriptors match the attributes of the arguments in the argument list at the point of invocation.

group. A collection of statements contained within larger program units. A group is either a do-group or a select-group and it can be used wherever a single statement can appear, except as an on-unit.

H

hex. See *hexadecimal digit*.

hexadecimal. Pertaining to a numbering system with a base of sixteen; valid numbers use the digits 0 through 9 and the characters A through F, where A represents 10 and F represents 15.

hexadecimal digit. One of the digits 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively.

I

identifier. A string of characters, not contained in a comment or constant, and preceded and followed by a delimiter. The first character of the identifier must be one of the 26 alphabetic characters and extralingual characters, if any. The other characters, if any, can additionally include extended alphabetic, digit, or the break character.

IEEE. Institute of Electrical and Electronics Engineers.

implicit. The action taken in the absence of an explicit specification.

implicit action. The action taken when an enabled condition is raised and no ON-unit is currently established for the condition. Contrast with *ON-statement action*.

implicit declaration. A name not explicitly declared in a DECLARE statement or contextually declared.

implicit opening. The opening of a file as the result of an input or output statement other than the OPEN statement.

infix operator. An operator that appears between two operands.

inherited dimensions. For a structure, union, or element, those dimensions that are derived from the containing structures. If the name is an element that is not an array, the dimensions consist entirely of its inherited dimensions. If the name is an element that is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure with one or more inherited dimensions is called a nonconnected aggregate. Contrast with *connected aggregate*.

input/output. The transfer of data between auxiliary medium and main storage.

insertion point character. A picture specification character that is, on assignment of the associated data to a character string, inserted in the indicated position. When used in a P-format item for input, the insertion character is used for checking purposes.

integer. An optionally signed sequence of digits or a sequence of bits without a decimal or binary point. An optionally signed whole number, commonly described as FIXED BINARY (p,0) or FIXED DECIMAL (p,0).

integral boundary. A byte multiple address of any 8-bit unit on which data can be aligned. It usually is a halfword, fullword, or doubleword (2-, 4-, or 8-byte multiple respectively) boundary.

interleaved array. An array that refers to nonconnected storage.

interleaved subscripts. Subscripts that exist in levels other than the lowest level of a subscripted qualified reference.

internal block. A block that is contained in another block.

internal name. A name that is known only within the block in which it is declared, and possibly within any contained blocks.

internal procedure. A procedure that is contained in another block. Contrast with *external procedure*.

interrupt. The redirection of the program's flow of control as the result of raising a condition or attention.

invocation. The activation of a procedure.

invoke. To activate a procedure.

invoked procedure. A procedure that has been activated.

invoking block. A block that activates a procedure.

iteration factor. In an INITIAL attribute specification, an expression that specifies the number of consecutive elements of an array that are to be initialized with the given value. In a format list, an expression that specifies the number of times a given format item or list of format items is to be used in succession.

iterative do-group. A do-group whose DO statement specifies a control variable and/or a WHILE or UNTIL option.

K

key. Data that identifies a record within a direct-access data set. See *source key* and *recorded key*.

keyword. An identifier that has a specific meaning in PL/I when used in a defined context.

keyword statement. A simple statement that begins with a keyword, indicating the function of the statement.

known (applied to a name). Recognized with its declared meaning. A name is known throughout its scope.

L

label. A name prefixed to a statement. A name on a PROCEDURE statement is called an entry constant; a name on a FORMAT statement is called a format constant; a name on other kinds of statements is called a label constant. A data item that has the LABEL attribute.

label constant. A name written as the label prefix of a statement (other than PROCEDURE, ENTRY, FORMAT, or PACKAGE) so that, during execution, program control can be transferred to that statement through a reference to its label prefix.

label data. A label constant or the value of a label variable.

label prefix. A label prefixed to a statement.

label variable. A variable declared with the LABEL attribute. Its value is a label constant in the program.

leading zeroes. Zeros that have no significance in an arithmetic value. All zeros to the left of the first nonzero in a number.

level number. A number that precedes a name in a DECLARE statement and specifies its relative position in the hierarchy of structure names.

level-one variable. A major structure or union name. Any unsubscripted variable not contained within a structure or union.

lexically. Relating to the left-to-right order of units.

library. An MVS partitioned data set or a CMS MACLIB that can be used to store other data sets called members.

list-directed. The type of stream-oriented transmission in which data in the stream appears as constants separated by blanks or commas and for which formatting is provided automatically.

locator. A control block that holds the address of a variable or its descriptor.

locator/descriptor. A locator followed by a descriptor. The locator holds the address of the variable, not the address of the descriptor.

locator qualification. In a reference to a based variable, either a locator variable or function reference connected by an arrow to the left of a based variable to specify the generation of the based variable to which the reference refers. It might be an implicit reference.

locator value. A value that identifies or can be used to identify the storage address.

locator variable. A variable whose value identifies the location in main storage of a variable or a buffer. It has the POINTER or OFFSET attribute.

locked record. A record in an EXCLUSIVE DIRECT UPDATE file that has been made available to one task only and cannot be accessed by other tasks until the task using it relinquishes it.

logical level (of a structure or union member). The depth indicated by a level number when all level numbers are in direct sequence (when the increment between successive level numbers is one).

logical operators. The bit-string operators not and exclusive-or (¬), and (&), and or (|).

loop. A sequence of instructions that is executed iteratively.

lower bound. The lower limit of an array dimension.

M

main procedure. An external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. This procedure is invoked automatically as the first step in the execution of a program.

major structure. A structure whose name is declared with level number 1.

member. A structure, union, or element name in a structure or union. Data sets in a library.

minor structure. A structure that is contained within another structure or union. The name of a minor structure is declared with a level number greater than one and greater than its parent structure or union.

mode (of arithmetic data). An attribute of arithmetic data. It is either *real* or *complex*.

multiple declaration. Two or more declarations of the same identifier internal to the same block without different qualifications. Two or more external declarations of the same identifier.

multiprocessing. The use of a computing system with two or more processing units to execute two or more programs simultaneously.

multiprogramming. The use of a computing system to execute more than one program concurrently, using a single processing unit.

multitasking. A facility that allows a program to execute more than one PL/I procedure simultaneously.

N

name. Any identifier that the user gives to a variable or to a constant. An identifier appearing in a context where it is not a keyword. Sometimes called a user-defined name.

nesting. The occurrence of:

- A block within another block
- A group within another group
- An IF statement in a THEN clause or in an ELSE clause
- A function reference as an argument of a function reference
- A remote format item in the format list of a FORMAT statement
- A parameter descriptor list in another parameter descriptor list
- An attribute specification within a parenthesized name list for which one or more attributes are being factored

nonconnected storage. Storage occupied by nonconnected data items. For example, interleaved arrays and structures with inherited dimensions are in nonconnected storage.

null locator value. A special locator value that cannot identify any location in internal storage. It gives a

positive indication that a locator variable does not currently identify any generation of data.

null statement. A statement that contains only the semicolon symbol (;). It indicates that no action is to be taken.

null string. A character, graphic, or bit string with a length of zero.

numeric-character data. See *decimal picture data*.

numeric picture data. Picture data that has an arithmetic value as well as a character value. This type of picture data cannot contain the characters 'A' or 'X.'

O

object. A collection of data referred to by a single name.

offset variable. A locator variable with the OFFSET attribute, whose value identifies a location in storage relative to the beginning of an area.

ON-condition. An occurrence, within a PL/I program, that could cause a program interrupt. It can be the detection of an unexpected error or of an occurrence that is expected, but at an unpredictable time.

ON-statement action. The action explicitly established for a condition that is executed when the condition is raised. When the ON-statement is encountered in the flow of control for the program, it executes, establishing the action for the condition. The action executes when the condition is raised if the ON-unit is still established or a RESIGNAL statement reestablishes it. Contrast with *implicit action*.

ON-unit. The specified action to be executed when the appropriate condition is raised.

opening (of a file). The association of a file with a data set.

operand. The value of an identifier, constant, or an expression to which an operator is applied, possibly in conjunction with another operand.

operational expression. An expression that consists of one or more operators.

operator. A symbol specifying an operation to be performed.

option. A specification in a statement that can be used to influence the execution or interpretation of the statement.

P

package constant. The label prefix on a PACKAGE statement.

packed decimal. The internal representation of a fixed-point decimal data item.

padding. One or more characters, graphics, or bits concatenated to the right of a string to extend the string to a required length. One or more bytes or bits inserted in a structure or union so that the following element within the structure or union is aligned on the appropriate integral boundary.

parameter. A name in the parameter list following the PROCEDURE statement, specifying an argument that will be passed when the procedure is invoked.

parameter descriptor. The set of attributes specified for a parameter in an ENTRY attribute specification.

parameter descriptor list. The list of all parameter descriptors in an ENTRY attribute specification.

parameter list. A parenthesized list of one or more parameters, separated by commas and following either the keyword PROCEDURE in a procedure statement or the keyword ENTRY in an ENTRY statement. The list corresponds to a list of arguments passed at invocation.

partially qualified name. A qualified name that is incomplete. It includes one or more, but not all, of the names in the hierarchical sequence above the structure or union member to which the name refers, as well as the name of the member itself.

picture data. Numeric data, character data, or a mix of both types, represented in character form.

picture specification. A data item that is described using the picture characters in a declaration with the PICTURE attribute or in a P-format item.

picture specification character. Any of the characters that can be used in a picture specification.

PL/I character set. A set of characters that has been defined to represent program elements in PL/I.

PL/I prompter. Command processor program for the PLI command that checks the operands and allocates the data sets required by the compiler.

point of invocation. The point in the invoking block at which the reference to the invoked procedure appears.

pointer. A type of variable that identifies a location in storage.

pointer value. A value that identifies the pointer type.

pointer variable. A locator variable with the POINTER attribute that contains a pointer value.

precision. The number of digits or bits contained in a fixed-point data item, or the minimum number of significant digits (excluding the exponent) maintained for a floating-point data item.

prefix. A label or a parenthesized list of one or more condition names included at the beginning of a statement.

prefix operator. An operator that precedes an operand and applies only to that operand. The prefix operators are plus (+), minus (−), and not (¬).

preprocessor. A program that examines the source program before the compilation takes place.

preprocessor statement. A special statement appearing in the source program that specifies the actions to be performed by the preprocessor. It is executed as it is encountered by the preprocessor.

primary entry point. The entry point identified by any of the names in the label list of the PROCEDURE statement.

priority. A value associated with a task, that specifies the precedence of the task relative to other tasks.

problem data. Coded arithmetic, bit, character, graphic, and picture data.

problem-state program. A program that operates in the problem state of the operating system. It does not contain input/output instructions or other privileged instructions.

procedure. A collection of statements, delimited by PROCEDURE and END statements. A procedure is a program or a part of a program, delimits the scope of names, and is activated by a reference to the procedure or one of its entry names. See also *external procedure* and *internal procedure*.

procedure reference. An entry constant or variable. It can be followed by an argument list. It can appear in a CALL statement or the CALL option, or as a function reference.

program. A set of one or more external procedures or packages. One of the external procedures must have the OPTIONS(MAIN) specification in its procedure statement.

program control data. Area, locator, label, format, entry, and file data that is used to control the processing of a PL/I program.

prologue. The processes that occur automatically on block activation.

pseudovariable. Any of the built-in function names that can be used to specify a target variable. It is usually on the left-hand side of an assignment statement.

Q

qualified name. A hierarchical sequence of names of structure or union members, connected by periods, used to identify a name within a structure. Any of the names can be subscripted.

R

range (of a default specification). A set of identifiers and/or parameter descriptors to which the attributes in a DEFAULT statement apply.

record. The logical unit of transmission in a record-oriented input or output operation. A collection of one or more related data items. The items usually have different data attributes and usually are described by a structure or union declaration.

recorded key. A character string identifying a record in a direct-access data set where the character string itself is also recorded as part of the data.

record-oriented data transmission. The transmission of data in the form of separate records. Contrast with *stream data transmission*.

recursive procedure. A procedure that can be called from within itself or from within another active procedure.

reentrant procedure. A procedure that can be activated by multiple tasks, threads, or processes simultaneously without causing any interference between these tasks, threads, and processes.

REFER expression. The expression preceding the keyword REFER, which is used as the bound, length, or size when the based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

REFER object. The variable in a REFER option that holds or will hold the current bound, length, or size for the member. The REFER object must be a member of the same structure or union. It must not be locator-qualified or subscripted, and it must precede the member with the REFER option.

reference. The appearance of a name, except in a context that causes explicit declaration.

relative virtual origin (RVO). The actual origin of an array minus the virtual origin of an array.

remote format item. The letter R followed by the label (enclosed in parentheses) of a FORMAT statement. The

format statement is used by edit-directed data transmission statements to control the format of data being transmitted.

repetition factor. A parenthesized unsigned integer constant that specifies:

1. The number of times the string constant that follows is to be repeated.
2. The number of times the picture character that follows is to be repeated.

repetitive specification. An element of a data list that specifies controlled iteration to transmit one or more data items, generally used in conjunction with arrays.

restricted expression. An expression that can be evaluated by the compiler during compilation, resulting in a constant. Operands of such an expression are constants, named constants, and restricted expressions.

returned value. The value returned by a function procedure.

RETURNS descriptor. A descriptor used in a RETURNS attribute, and in the RETURNS option of the PROCEDURE and ENTRY statements.

S

scalar variable. A variable that is not a structure, union, or array.

scale. A system of mathematical notation whose representation of an arithmetic value is either fixed-point or floating-point.

scale factor. A specification of the number of fractional digits in a fixed-point number.

scaling factor. See *scale factor*.

scope (of a condition prefix). The portion of a program throughout which a particular condition prefix applies.

scope (of a declaration or name). The portion of a program throughout which a particular name is known.

secondary entry point. An entry point identified by any of the names in the label list of an entry statement.

select-group. A sequence of statements delimited by SELECT and END statements.

selection clause. A WHEN or OTHERWISE clause of a select-group.

self-defining data. An aggregate that contains data items whose bounds, lengths, and sizes are determined at program execution time and are stored in a member of the aggregate.

separator. See *delimiter*.

shift. Change of data in storage to the left or to the right of original position.

shift-in. Symbol used to signal the compiler at the end of a double-byte string.

shift-out. Symbol used to signal the compiler at the beginning of a double-byte string.

sign and currency symbol characters. The picture specification characters. S, +, −, and $ (or other national currency symbols enclosed in < and >).

simple parameter. A parameter for which no storage class attribute is specified. It can represent an argument of any storage class, but only the current generation of a controlled argument.

simple statement. A statement other than IF, ON, WHEN, and OTHERWISE.

source. Data item to be converted for problem data.

source key. A key referred to in a record-oriented transmission statement that identifies a particular record within a direct-access data set.

source program. A program that serves as input to the source program processors and the compiler.

source variable. A variable whose value participates in some other operation, but is not modified by the operation. Contrast with *target variable*.

spill file. Data set named SYSUT1 that is used as a temporary workfile.

standard default. The alternative attribute or option assumed when none has been specified and there is no applicable DEFAULT statement.

standard file. A file assumed by PL/I in the absence of a FILE or STRING option in a GET or PUT statement. SYSIN is the standard input file and SYSPRINT is the standard output file.

standard system action. Action specified by the language to be taken for an enabled condition in the absence of an ON-unit for that condition.

statement. A PL/I statement, composed of keywords, delimiters, identifiers, operators, and constants, and terminated by a semicolon (;). Optionally, it can have a condition prefix list and a list of labels. See also *keyword statement*, *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.

T

target. Attributes to which a data item (source) is converted.

target reference. A reference that designates a receiving variable (or a portion of a receiving variable).

target variable. A variable to which a value is assigned.

task. The execution of one or more procedures by a single flow of control.

task name. An identifier used to refer to a task variable.

task variable. A variable with the TASK attribute whose value gives the relative priority of a task.

termination (of a block). Cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or to some other active block by means of a GO TO statement.

termination (of a task). Cessation of the flow of control for a task.

truncation. The removal of one or more digits, characters, graphics, or bits from one end of an item of data when a string length or precision of a target variable has been exceeded.

type. The set of data attributes and storage attributes that apply to a generation, a value, or an item of data.

U

undefined. Indicates something that a user must not do. Use of a undefined feature is likely to produce different results on different implementations of a PL/I product. In that case, the application program is in error.

union. A collection of data elements that overlay each other, occupying the same storage. The members can be structures, unions, elementary variables, or arrays. They need not have identical attributes.

union of arrays. A union that has the DIMENSION attribute.

upper bound. The upper limit of an array dimension.

V

value reference. A reference used to obtain the value of an item of data.

variable. A named entity used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times.

variable reference. A reference that designates all or part of a variable.

virtual origin (VO). The location where the element of the array whose subscripts are all zero are held. If such an element does not appear in the array, the virtual origin is where it would be held.

Z

zero-suppression characters. The picture specification characters Z and *, which are used to suppress zeros in the corresponding digit positions and replace them with blanks or asterisks respectively.

Index

Special characters

/ (forward slash) 162
*PROCESS statement 24
%INCLUDE statement 24
%LINE directive 25
%OPTION directive 25
%PROCESS statement
  and PROCEDURE statement 24
  specifying compile-time options 37

A

access methods
  DDM 144
  I/O 145
accessing data sets
  examples of REGIONAL(1) 186
  record I/O 185
  REGIONAL(1) 197
  stream I/O 173
adapting existing programs for workstation VSAM 206
ADDBUFF ENVIRONMENT option 10
aggregate
  length table, example 120
AGGREGATE compile-time option 44
AIX
  device names 23
  export command 23
  features
    file specification 22
  setting environment variables 23
AIX devices 144
AIX operating system
  configuration file 31
  environment variables 30
  flags 36
  input files 34
AIX_filespec 153
ALIGNED compile-time suboption 54
alternate ddname, in TITLE option 162
American National Standard (ANS)
  in CTLASA option 148
  in printer-destined files 169
AMTHD option 153
ANS
  compile-time suboption 50
  control character 148, 169
  print control characters 170
APPEND option 154
AREAs and INITIAL attribute 17
array expressions restrictions 11
ASA option 155
ASCII
  compile-time suboption
    description 50
    effect on performance 241
  data conversion tables 234
  DBCS portability 16
  portability considerations 14
ASCII ENVIRONMENT option 10
ASSIGNABLE compile-time suboption 50
attributes and cross-reference table 119
avoiding calls to library routines 247

B

BACKWARDS file attribute 10
base file of VSAM keyed data set 202
BKWD option 147
BOOKSHELF environment variable 31
Bourne shell 23, 30
BUFFERS ENVIRONMENT option 10
BUFND ENVIRONMENT option 10
BUFNI ENVIRONMENT option 10
BUFOFF ENVIRONMENT option 11
BUFSIZE option 155
built-in functions
  restricted 12
BYADDR
  description 240
  effect on performance 240
  using with DEFAULT option 50
byte-reversed integers 15
BYVALUE
  description 240
  effect on performance 240
  using with DEFAULT option 50

C

C shell 23, 30
carriage return-line feed (CR - LF) 160
case sensitivity 10
CHECK compile-time option 46
CICS
  preprocessor options 108
  support 107
CMPAT compile-time option 46
code inspection 125
coding
  CICS statements 109
  embedded control characters 9
  improving performance 242
  SQL statements 93
command line
  specifying compile-time options 35
communications area, SQL 93
compatibility of OS PL/I files for the workstation 206
compilation
  compile-time options 41
  failure 139
  mainframe applications on your workstation 9
  preparing your source program 24
  sample output 111
  user exit
    activating 230
compilation *(continued)*
  user exit *(continued)*
    customizing 231
    IBMUEXIT 230
    procedures 229
  using the PLI command to invoke the compiler 29
compilation environment 30
compilation output
  compiler output 121
  using the compiler listing 111
compile-time options 10
  AGGREGATE 44
  CHECK 46
  CMPAT 46
  DEFAULT 239
  GONUMBER 238
  IMPRECISE 238
  MAXSTMT 63
  MDECK 64
  NATLANG 65
  NUMBER 66
  OPTIMIZE 237
  PP 68
  PPTRACE 69
  PREFIX 239
  PROBE 70
  RULES 238
  specifying 35
  SYSTEM 78
  USAGE 79
  use in debugging 127
  using the configuration file 31
  WIDECHAR 79
  WINDOW 80
  XINFO 80
compiler
  descriptions of options 41
  listing
    stack storage used 77
compiler options
  abbreviations 42
  COPYRIGHT 48
  default 41
  EXTRN 54
  REDUCE 71
  STATIC 76
  STORAGE 77
  XINFO 80
compiler restrictions
  array expressions 11
  built-in functions 12
  DBCS 12
  DEFAULT statement 11
  extents of automatic variables 12
  iSUB defining 12
  MACRO preprocessor 12
  pseudovariables 12
  RECORD I/O 10
  STREAM I/O 11
  structure expressions 11

concatenation 26
condition handling
coding ON-units 135
general concepts 133
interrupts 133
list of conditions and their attributes 136
qualified and unqualified conditions 136
scope and descendancy 133
terminology 133
conditions
handling conversions inline 248
handling string built-in functions inline 248
configuration file
attributes
library options 32
linkage options 32
path name for object file 32
preprocessor options 32
example 32
include preprocessor 84
macro facility 86
specifying compile-time options 35
SQL preprocessor 92
tailoring 31
CONNECT TO statement 104
CONNECTED compile-time suboption
description 50
effect on performance 241
CONSECUTIVE
files 206
option
definition 147
stream I/O 171
consecutive data sets
controlling input from the console
capital and lowercase letters 182
end of file 182
format of data 181
stream and record files 181
using files conversationally 181
controlling output to the console
example of an interactive program 182
format of PRINT files 182
stream and record files 182
description 169
examples 186
PRINT files 182
printer-destined files 169
using record-oriented I/O
accessing and updating a data set 185
creating a data set 185
defining files 184
ENVIRONMENT options for data transmission 185
using stream-oriented data transmission
accessing a data set with stream I/O 173
creating a data set with stream I/O 171
defining files using stream I/O 171
consecutive data sets *(continued)*
using stream-oriented data transmission *(continued)*
ENVIRONMENT options for 171
using PRINT files 175
using SYSIN and SYSPRINT files 179
console
input 180
output 182
control blocks
function-specific 230
global control 232
control characters
ANS in CTLASA option 148
printer 169
conversion tables 234
COPYRIGHT compiler option 48
cross-reference table in compilation output 119
CTLASA option 148
CURRENCY compile-time option
portability 9
customizing
user exit
modifying IBMUEXIT.INF 231
structure of global control blocks 232
writing your own compiler exit 232

D

data
conversion 162
conversion tables 234
files
associating a data file with OPEN 163
closing a PL/I file 163
creating 163
record 146
representations, portability 14
testing 126
transmission 147
types
equivalent SQL and PL/I 96
data sets
access methods 145
accessing
examples of REGIONAL(1) 186
record I/O 185
associating a PL/I file with a data set
how PL/I finds data sets 163
using environment variables 162
using the TITLE option of the OPEN statement 162
using unassociated files 163
associating several data sets with one file 164
associating with more than one file 163
characteristics 143
combinations of I/O statements, attributes, and options 164
DD_DDNAME environment variable 161
data sets *(continued)*
DD:ddname environment variable 152
default identification 161
defining and using 202
disassociating 163
DISPLAY statement input and output 166
establishing a path 163
establishing characteristics
data set organizations 146
DD_DDNAME environment variable 153
PL/I ENVIRONMENT attribute 147
record formats 146
records 146
extending on output 154
keyed access 145
maximum number of regions 157
native, fixed-length 144
number of regions 157
opening a PL/I file 163
organization
DDM and VSAM 153
default 146
options 146
regional 151
PL/I standard files (SYSPRINT and SYSIN) 167
recreating output 154
redirecting standard input output, and error devices 167
regional 145
REGIONAL(1) 145
specifying characteristics 146
stream files 170
types
conventional text files and devices 144
fixed-length data sets 144
regional data sets 145
VSAM 145
workstation VSAM
defining files 202
direct data sets 218
keyed data sets 211
organization 202
sequential data sets 208
data-directed I/O
coding for performance 243
DBCS constants 150
specifying GRAPHIC option 150
DBCS (double-byte character set)
and GRAPHIC option 150
DBCS restrictions 12
DD information
record format 146
TITLE statement 162
DD_DDNAME environment variables
alternate ddname 162
DD:ddname environment variables 152
specifying characteristics 152
DDM access method 144, 145
DDM data sets
record formats 146
value of AMTHD 153

debugging programs
common PL/I errors
compiler or library subroutine failure 139
invalid use of PL/I 137
logical errors in source 136
loops and other unforeseen errors 137
poor performance 140
system failure 140
unexpected input/output data 138
unexpected program results 139
unexpected program termination 138
uninitialized entry variables 137
condition handling 133
dumps 129
FLAG option 127
general debugging tips 126
GONUMBER option 127
NOLAXDCL option 128
NOLAXIF option 128
PREFIX option 128
RULES option 128
using compile-time options 127
using footprints for debugging
DISPLAY 129
PUT DATA 129
PUT LIST 129
PUT SKIP LIST 129
using the Distributed Debugger tool 127
XREF option 128
DECLARE
STATEMENT definition 105
TABLE statement 105
declaring
host variables, SQL preprocessor 95
DEFAULT compile-time option
suboptions
ALIGNED 54
ASCII or EBCDIC 50
ASSIGNABLE or NONASSIGNABLE 50
BYADDR or BYVALUE 50
CONNECTED or NONCONNECTED 50
DESCLIST or DESCLOCATOR 52
DESCRIPTOR or NODESCRIPTOR 50
DUMMY 53
E 54
EVENDEC or NOEVENDEC 52
IBM or ANS 50
IEEE or HEXADEC 51
INITFILL or NOINITFILL 52
INLINE or NOINLINE 51
LOWERINC or UPPERINC 52
NATIVE or NONNATIVE 51
NATIVEADDR or NONNATIVEADDR 51
NOFROMALIEN or FROMALIEN 52
NULLSYS or NULL370 52
ORDER or REORDER 51
ORDINAL 51
DEFAULT compile-time option *(continued)*
suboptions *(continued)*
OVERLAP or NOOVERLAP 51
RECURSVIE or NONRECURSIVE 52
RETCODE 53
RETURNS 53
SHORT 53
using default suboptions 239
DEFAULT statement restrictions 11
DEFINED
versus UNION 246
defining files
for data sets 204
for REGIONAL(1) data sets 193
defining library paths 31
DELAY option
description and syntax 156
DELETE statement 165
DELIMIT option
description and syntax 156
DESCLIST compile-time suboption 52
DESCLOCATOR compile-time suboption 52
descriptor area, SQL 93
DESCRIPTOR compile-time option
effect on performance 241
DESCRIPTOR compile-time suboption
description 50
desk checking 125
device
character 143
con 166
names 23
standard 143
std 166
direct access 195
direct data sets 218, 227
DIRECT file
using to access a workstation VSAM direct data set 223
using to access a workstation VSAM keyed data set 215
directing I/O 166
DISPLAY 129
Distributed Data Management 143
Distributed Debugger debug tool 127
DSNTIAR.PLI sample program 105
dummy
devices (AIX) 144
records 191
DUMMY compile-time suboption 53
dumps
condition handling 133
default options 131
error handling 133
formatted PL/I dumps—PLIDUMP 131
options string 130
SNAP dumps for trace information 133
title string 130
dynamic descendancy 133

E

E compile-time suboption 54
EBCDIC
compile-time suboption 50
data conversion tables 234
DBCS portability 16
effect on performance 241
portability considerations 14
edit-directed I/O 150
embedded
CICS statements 109
control characters 9
SQL statements 94
ENCINA access method 153
Encina file access 201
end of file characters (/*) 182
ENVIRONEMENT options not supported 10
ENVIRONMENT attribute
(REREAD) on the CLOSE statement
regional data sets 193
options
CTLASA 169
specifying characteristics 147
BKWD 147
BUFSIZE 155
CONSECUTIVE 147
CTLASA 148
GENKEY 148
GRAPHIC 150
KEYLENGTH 150
KEYLOC 150
ORGANIZATION 151
RECSIZE 151
REGIONAL(1) 151
SCALARVARYING 152
VSAM 152
specifying options
for record I/O 185
for workstation VSAM data sets 205
stream I/O 171
environment differences, S/390 and AIX 16
ENVIRONMENT options
for record-oriented data transmission
CONSECUTIVE 185
CTLASA 185
ORGANIZATION(CONSECUTIVE) 185
RECSIZE 185
SCALARVARYING 185
stream-oriented data transmission 171
environment variables
Bourne shell 30
C shell 30
defining library paths 31
Korn shell 30
locating online documentation 31
ERROR
ON-units 18
error and condition handling
conditions used for testing and debugging 136
dynamic descendancy 133
general concepts 133
interrupts and PL/I conditions 133

error and condition handling *(continued)*
normal return 134
ON-units for conditions 136
standard system action 134
static descendancy 133
terminology 133
errors
calling uninitialized entry variables 137
compiler or library 139
differences in issuing from OS PL/I 18
invalid use of PL/I 137
logical errors in source program 136
loops 137
poor performance 140
run-time messages 135
system failure 140
unexpected
input/output data 138
program results 139
program termination 138
unforeseen errors 137
EVENDEC compile-time suboption 52
EXCLUSIVE file attribute 10
EXEC SQL statements 87
export command 23
EXTRN compiler option 54

F

features of PL/I for AIX 21
file
specification 22
file attributes not supported
BACKWARDS 10
EXCLUSIVE 10
TRANSIENT 10
filenames, case sensitivity 10
files
adapting existing programs for workstation VSAM
using CONSECUTIVE files 206
using INDEXED files 206
using REGIONAL(1) files 206
using VSAM files 206
closing 163
declarations for REGIONAL(1) data sets 191, 192, 193
defining
record I/O 184
stream I/O 171
opening 163
PL/I
definition 143
standard 167
printer-destined 170
STREAM attribute 170
SYSIN 179
SYSPRINT 179
FILLERS 179
filtering messages 230
FIXED
BINARY, mapping and portability 17
TYPE option 160
fixed-length record format 146
FLAG compile-time option
using when debugging 127
flags, specifying compile-time options 36
floating-point data 15
footprints for debugging 128
formatted PL/I dumps 130
FROMALIEN compile-time suboption 52, 256

G

GENKEY option 147, 148
GET statement 165
GET statements
controlling input from the console 180
GRAPHIC option 150
global control blocks
data entry fields 232
writing the initialization procedure 233
writing the message filtering procedure 233
writing the termination procedure 234
GONUMBER compile-time option 238
using when debugging 127
GRAPHIC
ENVIRONMENT option 150, 171
graphic data and stream I/O 170

H

handling conditions
built-in for condition handling 135
PL/I run-time error message formats 135
SNAP messages 135
system messages 135
sources of conditions 135
hello program 22
HEXADECIMAL
compile-time suboption 51
portability considerations 15
host
structures 102
variables, using in SQL statements 95

I

I/O
access methods
BTRIEVE 145
ENCINA 153
ISAM 145
attributes table 164
DDM 145
options table 164
redirection 167
statements table 164
unexpected 138
using the sort program 266
IBM compile-time suboption 50
IBMUEXIT compiler exit 230
IEEE
compile-time suboption 51
portability considerations 15
IMPRECISE compile-time option
improving performance 238
improving application performance 237
include preprocessor
syntax 84
using configuration file 84
INCLUDE processing 10, 24
INDEXAREA ENVIRONMENT option 11
indexed data sets 145
INDEXED files, adapting programs 206
indicator variables, SQL 103
INITFILL compile-time suboption 52
INITIAL attribute 17
initialization procedure of compiler user exit 233
INLINE compile-time suboption 51
input
controlling from console 180
defining data sets for stream files 171
example of interactive program 182
files 34
SEQUENTIAL 183
to the console
example of an interactive program 182
format of PRINT files 182
stream and record files 182
input and output with workstation VSAM data sets
description 201
organization
accessing records in 202
creating and accessing 202
determining which type you need 202
using keys 203
using workstation VSAM direct data sets
loading 220
using a DIRECT file to access 223
using a SEQUENTIAL file to access 222
using workstation VSAM keyed data sets
loading 213
using a DIRECT file to access 215
using a SEQUENTIAL file to access 215
using workstation VSAM sequential data sets
defining and loading 209
updating 210
using a SEQUENTIAL file to access 209
input- and output-handling routines, sort program 266
installation 307
interactive program, example 182
interlanguage communication (ILC) 251
interrupts 128
invoking
compiler 29

invoking *(continued)*
linkage editor 38
ISAM access method 145
iSUB defining restrictions 12

K

key
accessing a sequential data set 204
generic 148
relative record number
padding 204
truncation 204
starting position 150
KEY
keys for workstation VSAM keyed data sets 203
option in READ statement 147
relative record numbers 204
sequential record values 204
key length, checking 150
keyboard
screen operations 166
keyed data sets 215
statements and options for 211
types and advantages 203
KEYFROM 203
KEYFROM, relative record numbers 204
KEYLENGTH option 150
KEYLOC option 150
keys
using for workstation VSAM keyed data sets 203
using relative record numbers 204
using sequential record values 204
KEYTO
keys for workstation VSAM keyed data sets 203
relative record numbers 204
sequential file to access a workstation VSAM sequential data set 209
sequential record values 204
Korn shell 23, 30

L

large object (LOB) support, SQL preprocessor 98
LEAVE ENVIRONMENT option 11
length of record
maximum 151
specifying 157
LIBPATH environment variable 31
library
files 34
options attribute 32
library, compiler subroutine failure 139
line continuation 26
line feed (LF)
definition 160
delimiting logical records 144
LF files 146
LINE option
in controlling output to the console 182
of PUT statement 169
LINE option *(continued)*
using with PRINT files 175
when using PRINT files 175
LINESIZE option
accessing a data set with stream I/O 171
creating a data set with stream I/O 171
definition 176
OPEN statement 171
tab set table field 179
linkage editor
invoking 38
options attribute 32
list-directed I/O
DBCS constants 150
specifying GRAPHIC option 150
LOCATE statement 165
locating online documentation 31
logical errors in source 136
loops
coding ON-units 137
control variables 244
tips for use 137
LOWERINC compile-time suboption 52
LRECL option 156
LRMSKIP option 156

M

machine interrupts 133
macro facility
macro definition 85
portability 13
using configuration file 86
macro preprocessor
macro definition 85
mainframe applications
running on the workstation 14
margins in source program 26
MAXSTMT compile-time option 63
MDECK compile-time option 64
messages
filter function 233
modifying in compiler user exit 231
migration
compatibility with OS PL/I 9
OS PL/I files for the workstation
CONSECUTIVE file 206
EXCLUSIVE file 205
INDEXED file 206
ISAM record handling 206
REGIONAL(1) file 206
VSAM file 206
mixed-language applications 251
module testing 126

N

named constants
defining 246
versus static variables 246
national characters 9
NATIVE compile-time suboption
description 51
effect on performance 241
NATIVE compile-time suboption *(continued)*
portability considerations 15
native data sets
character devices 144
conventional text files 144
DDM data sets 145
fixed-length data sets 144
regional data sets 145
NATIVEADDR compile-time suboption 51
NATLANG
compile-time option 65
NCP ENVIRONMENT option 11
NODESCRIPTOR compile-time suboption 50
NOEVENDEC compile-time suboption 52
NOFROMALIEN compile-time suboption 52
NOINITFILL compile-time suboption 52
NOINLINE compile-time suboption 51
NOLAXDCL compile-time option 128
NOLAXIF compile-time option 128
NONASSIGNABLE compile-time suboption 50
NONCONNECTED compile-time suboption 50
NONNATIVE compile-time suboption 51
NONNATIVEADDR compile-time suboption 51
NONRECURSIVE compile-time suboption 52
NOOVERLAP compile-time suboption
description 51
NOT compile-time option
portability 9
notices 309
NOWRITE ENVIRONMENT option 11
NULL370 compile-time suboption 52
NULLSYS compile-time suboption 52
NUMBER compile-time option 66, 119

O

object files 34
offset
determining statement numbers 135
tab count 178
online documentation 31
OPEN statement
opening a file 163
specifying the length of records 146
using
LINESIZE option 171
TITLE option 146, 163
Operating system
data definition (DD) information 162
optimal coding
coding style 242
compile-time options 237
OPTIMIZE compile-time option 237
options
compile-time
specifying 35

options *(continued)*
DD_DDNAME environment variables
AMTHD 153
APPEND 154
ASA 155
DELAY 156
DELIMIT 156
LRECL 156
LRMSKIP 156
PROMPT 157
PUTPAGE 157
RECCOUNT 157
RECSIZE 157
RETRY 158
SAMELINE 158
SHARE 159
SKIP0 159
TYPE 159
FROMALIEN 256
I/O table 164
PL/I ENVIRONMENT attribute
BKWD 147
BUFSIZE 155
CONSECUTIVE 147
CTLASA 148
GENKEY 148
GRAPHIC 150
KEYLENGTH 150
KEYLOC 150
ORGANIZATION(CONSECUTIVE) 151
ORGANIZATION(INDEXED) 151
ORGANIZATION(RELATIVE) 151
RECSIZE 151
REGIONAL(1) 151
SCALARVARYING 152
VSAM 152
PRINT attribute
LINE 175
PAGE 175
SKIP 175
using
DD information 162
TITLE 162
OR compile-time option
portability 9
ORDER compile-time suboption
description 51
effect on performance 241
ORDINAL compile-time suboption 51
organization
data sets 146
default 147
regional data sets 151
VSAM 152
ORGANIZATION option 151
output
defining data sets for stream files 171
SEQUENTIAL 183
to the console
example of an interactive program 182
format of PRINT files 182
stream and record files 182
OVERLAP compile-time suboption
description 51

P

PACKAGEs versus nested PROCEDUREs 244
page
PAGELENGTH tab set table field 179
PAGESIZE tab set table field 179
PAGE option
of PUT statement 169
using with PRINT files 175
path testing 126
performance improvement
coding for performance
avoiding calls to library routines 247
DATA-directed input and output 243
DEFINED versus UNION 246
loop control variables 244
named constants versus static variables 246
PACKAGEs versus nested PROCEDUREs 244
REDUCIBLE functions 245
selecting compile-time options
DEFAULT 239
GONUMBER 238
IMPRECISE 238
OPTIMIZE 237
PREFIX 239
RULES 238
PL/I
AIX compiler
system requirements 307
compiler
invalid language use 137
user exit procedures 230
ENVIRONMENT attribute
OPEN statement 146
options portable to other SAA implementations 147
features
summary of characteristics 21
file format 26
file structure 24
files
associating with a data set 161
definition 143
standard files 167
platform
differences 16
PLI command
invoking the compiler 29
PLIDUMP
discussion 130
obtaining
file information 130
TCA information 130
reading a formatted PL/I dump 132
suggested coding 131
PLISRTx
calling the sort program 264
communicating success or failure 261, 265
determining which subroutine to use 260
PLISRTx *(continued)*
input- and output-handling routines 266
parameters 259
sort data input and output 266
specifying the sorting field 263
PLITABS 179
poor performance 140
portability
avoiding logic errors 14
changes in run-time behavior 14
data representations 14
embedded control characters 9
environment differences 16
language elements 17
national characters and other symbols 9
operating system differences 9
using the macro facility 13
PP compile-time option 68
PPTRACE compile-time option 69
practice exercise 22
PREFIX compile-time option 239
using default suboptions 239
using when debugging 128
preparing your source program for compilation
%LINE directive 25
%OPTION directive 25
INCLUDE processing 24
line continuation 26
margins 26
program file format 26
program file structure 24
preprocessor options attribute 32
preprocessors
available with PL/I 83
CICS options 108
include 84
macro facility 85
macro preprocessor 85
SQL options 87
SQL preprocessor 87
PRINT files
applying the PRINT attribute 175
controlling printed line length 176
format at terminal 182
inserting ANS print control characters 175
overriding the tab control table 178
printer control character, ASA 169
printer-destined files 169
ANS print control characters
IBM Proprinter equivalents 170
list 170
ASA option 169
controlling printed line length 176
example of creating a file 178
overriding the tab control table 178
print control characters 169
PROMPT option 157
Proprinter, IBM, control characters 148
pseudovariables restricted 12
PUT
DATA 129
LIST 129
SKIP LIST 129

PUT *(continued)*
statement
attributes and options 164
controlling input from the console 180
GRAPHIC option 150
without FILE option 179
PUT statement
PAGE, SKIP, and LINE options 169
PUTPAGE option 157

R

READ statement, attributes, and options 165
RECCOUNT option 157
RECORD condition
adapting CONSECUTIVE file for workstation VSAM 206
adapting INDEXED file for workstation VSAM 206
RECORD file 150
record formats 146
RECORD I/O
restrictions 10
RECORD OUTPUT files
associated with consecutive data sets 148
using CTLASA 148
record-oriented I/O
accessing a data set 185
creating a data set 185
defining files using 184
ENVIRONMENT options for data transmission 185
essential information 186
examples of consecutive data sets 186
updating a data set with 185
records
accessing in workstation VSAM data sets 202
length 157
specifying length 146
RECSIZE option
description and syntax 157
for stream I/O 171
PL/I ENVIRONMENT attribute 151
specifying the length of records 146
RECURSIVE compile-time suboption 52
REDUCE compiler option 71
REDUCIBLE functions 245
region numbers 195
regional data sets
commands and options 191, 192, 193
description 191
file definition
specifying ENVIRONMENT options 193
using keys with regional data sets 194
required information 194
using REGIONAL(1) data sets
direct access 198
dummy records 194
example 198
sequential access 197
regional data sets *(continued)*
using REGIONAL(1) data sets *(continued)*
updating 197
REGIONAL ENVIRONMENT option 11
REGIONAL(1)
data sets
accessing and updating 197
creating 195
discussion 191
example 195
using direct access 198
using sequential access 197
ENVIRONMENT option 151
files 206
regions 157
relative record numbers in workstation VSAM data sets 204
remote file access 201
REORDER compile-time suboption
description 51
effect on performance 241
requirements, system 307
REREAD ENVIRONMENT option 11
RETCODE compile-time suboption 53
RETRY option 158
RETURNS compile-time suboption 53, 241
REWRITE statement 165
REWRITE statement, attributes and options 166
routines, library, conversions 248
RULES compile-time option
effect on performance 238
using when debugging 128
run-time
behavior differences
ERROR message issuing 18
INITIAL attribute for AREAs is ignored 17
language elements 17
using variables declared as FIXED BIN 17
differences between platforms 14
messages
SNAP 135
SYSTEM 135
running a program 38

S

SAA suboption of LANGLVL 59
SAA2 suboption of LANGLVL 59
SAMELINE option 158
sample programs 22
SCALARVARYING option 152
screen and keyboard operations 166
sequential
access 195
data sets
statements and options 208
workstation VSAM 203
record value
in workstation VSAM sequential data set 204
using KEYTO to find 209
SEQUENTIAL
INPUT 147
OUTPUT 184
UPDATE 147
SEQUENTIAL file
using to access a workstation VSAM direct data set 222
using to access a workstation VSAM keyed data set 215
using to access a workstation VSAM sequential data set 209
setting
compilation environment 30
environment variables 23
SHARE option 159
SHORT compile-time suboption 53
SIS ENVIRONMENT option 11
SKIP ENVIRONMENT option 11
SKIP option
controlling input from the console 181
of PUT statement 169
using with PRINT files 175
SKIP0 option 159
SNAP compile-time option
messages 135
sort exit
E15 266
E35 269
sort program
calling the sort program 264
communicating success or failure 261, 265
comparing S/390 to the workstation 259
input- and output-handling routines 266
PLISRTx 259
preparing to use sort 260
sort data input and output 266
specifying the sorting field 263
varying-length records 266
source key 194
source program
%LINE directive 25
%OPTION directive 25
as input 34
concatenation 26
file format 26
file structure 24
INCLUDE processing 24
line continuation 26
margins 26
specifying compile-time options
%PROCESS statement 37
command line 35
configuration file 35
how to 35
using flags 36
SQL preprocessor
communications area 93
descriptor area 93
error return codes, handling 105
EXEC SQL statements 87
large object support 98
options 87
user defined functions 100

SQL preprocessor *(continued)*
using configuration file 92
using host structures 102
using host variables 95
using indicator variables 103
SQLCA 93
SQLDA 93
standard
device, workstation 143
error device, AIX 144
system action 134
statement numbers, determining from offset 135
statements
DELETE 165
GET 165
LOCATE 165
READ 165
REWRITE 165
WRITE 165
STATIC compiler option 76
static descendancy 133
storage
report in listing 77
STORAGE compiler option 77
stream and record files 182
STREAM attribute
data sets 170
discussion 170
stream I/O
accessing data sets 173
creating a data set 171
essential information 171
example 172
STREAM I/O
restrictions 11
stream-oriented data transmission
accessing a data set with stream I/O
essential information 174
example 174
creating a data set with stream I/O 171
defining files using stream I/O 171
ENVIRONMENT options for stream-oriented data transmission 171
using PRINT files 175
using SYSIN and SYSPRINT files 179
structure expression restrictions 11
structure of global control blocks
writing the initialization procedure 233
writing the message filtering procedure 233
writing the termination procedure 234
SYSIN files
attributes 179
redirecting standard input 167
SYSPRINT files
attributes 179
redirecting standard output 167
system
error-handling facilities 134
failure 140
messages 135
requirements 307
system *(continued)*
standard action for conditions 134
SYSTEM
compile-time option 78
message 135

T

tab control table 178
terminal
conversational I/O 181
example of an interactive program 182
input 180
output 182
termination procedure
compiler user exit 234
example of procedure-specific control block 234
syntax
global 232
specific 234
testing programs
code inspection 125
data testing 126
path testing 126
text file
conventional 143
LF 143
TITLE option
description 162
opening and closing a file 163
specifying the length of records 146
using
RECSIZE option 171
SYSPRINT and SYSIN files 167
using files not associated with data sets 163
TOTAL ENVIRONMENT option 11
TP ENVIRONMENT option 11
trace information 130
TRANSIENT file attribute 10
TRKOFL ENVIRONMENT option 11
TYPE option 159
specifying record formats 146

U

U-format record 146
undefined-length record format 146
UNDEFINEDFILE condition
raising when opening a file 161
using files not associated with data sets 161
unexpected
input/output data 138
program end 138, 139
uninitialized entry variables 137
UNLOCK statement 10
UPPERINC compile-time suboption 52
USAGE compile-time option 79
user defined functions, SQL preprocessor 100
user exit
compiler 229
user exit *(continued)*
customizing
modifying IBMUEXIT.INF 231
structure of global control blocks 232
writing your own compiler exit 232
functions 230
using host variables, SQL preprocessor 95
using the sort program 259

V

varying-length records
format 146
sorting 266
VSAM
files, adapting programs 206
option 152

W

WIDECHAR compile-time option 79
WINDOW compile-time option 80
workstation
native data sets 143
record format 146
workstation VSAM data sets
accessing records 202
adapting programs
using CONSECUTIVE files 206
using INDEXED files 206
using REGIONAL(1) files 206
using VSAM files 206
choosing a type 204
defining files
adapting existing programs 205
specifying options of the PL/I ENVIRONMENT attribute 205
direct 202
file declaration 204
keyed
base file 202
prime index file 202
sequential 202
types and advantages 202
workstation VSAM direct data sets
loading 220
using a DIRECT file to access 223
using a SEQUENTIAL file to access 222
workstation VSAM keyed data sets
loading 213
using a DIRECT file to access 215
using a SEQUENTIAL file to access 215
workstation VSAM sequential data sets
accessing from a SEQUENTIAL file 209
defining and loading 209
updating 210
using a SEQUENTIAL file to access 209
WRITE statement, attributes and options 165

X

XINFO compile-time option 80
XINFO compiler option 80
XREF compile-time option
  output in listing 119
  using when debugging 128

Program Number: 5724-H45

Printed in USA

IBM PL/I for AIX Library

GC18-9327
: Install Guide

SC18-9328
: Programming Guide

SC27-1460
: Language Reference

SC27-1461
: Compile-Time Messages and Codes

SC18-9328-00