Enterprise PL/I for z/OS

Programming Guide

Version 3 Release 7

IBM

Enterprise PL/I for z/OS

Programming Guide

Version 3 Release 7

> **Note!**
>
> Before using this information and the product it supports, be sure to read the general information under "Notices" on page 443.

Contents

Tables

Figures

Introduction

About This Book

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

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

Run-time environment for Enterprise PL/I for z/OS

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

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

Using your documentation

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

The following tables show you how to use the publications you receive with Enterprise PL/I and Language Environment. You'll want to know information

about both your compiler and run-time environment. For the complete titles and order numbers of these and other related publications, see "Bibliography" on page 447.

PL/I information

Table 1. How to use Enterprise PL/I publications

To...	Use...
Evaluate Enterprise PL/I	Fact Sheet
Understand warranty information	Licensed Programming Specifications
Plan for and install Enterprise PL/I	Enterprise PL/I Program Directory
Understand compiler and run-time changes and adapt programs to Enterprise PL/I and Language Environment	Compiler and Run-Time Migration Guide
Prepare and test your programs and get details on compiler options	Programming Guide
Get details on PL/I syntax and specifications of language elements	Language Reference
Diagnose compiler problems and report them to IBM	Diagnosis Guide
Get details on compile-time messages	Compile-Time Messages and Codes

Language Environment information

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

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

Notation conventions used in this book

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

Conventions used

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

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

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

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

How to read the syntax notation

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.

Conventions

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

Required items

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

```
►►──REQUIRED_ITEM──────────────────────────────────────────────────►◄
```

Optional Items

Optional items appear below the main path.

```
►►──REQUIRED_ITEM──┬──────────────────┬──────────────────────────────►◄
                   └─optional_item────┘
```
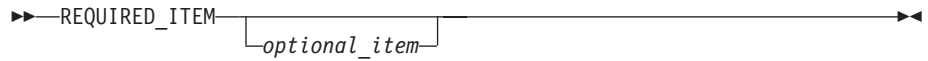
If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

```
                   ┌─optional_item────┐
►►──REQUIRED_ITEM──┴──────────────────┴──────────────────────────────►◄
```

Multiple required or optional items

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

```
►►──REQUIRED_ITEM──┬─required_choice1─┬──────────────────────────────►◄
                   └─required_choice2─┘
```
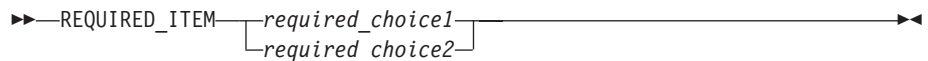
If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──REQUIRED_ITEM──┬──────────────────┬──────────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```
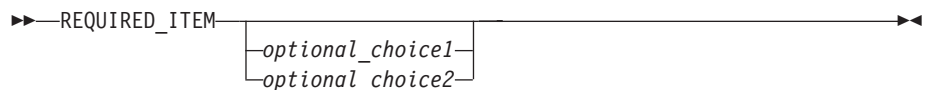
Repeatable items

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

```
                    ┌──────────────┐
►►──REQUIRED_ITEM───▼─repeatable_item─┴───────────────────────────────►◄
```
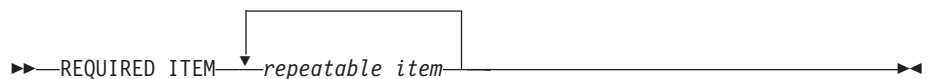
If the repeat arrow contains a comma, you must separate repeated items with a comma.

```
                    ┌──,───────────┐
►►──REQUIRED_ITEM───▼─repeatable_item─┴───────────────────────────────►◄
```
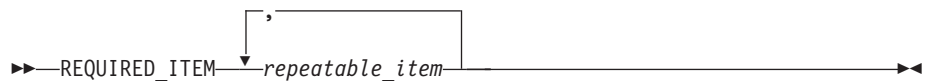
A repeat arrow above a stack indicates that you can specify more than one of the choices in the stack.

Default keywords

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

```
                    ┌─default_choice─┐
►►──REQUIRED_ITEM───┼────────────────┼──────────────────────────────────►◄
                    ├─optional_choice─┤
                    └─optional_choice─┘
```
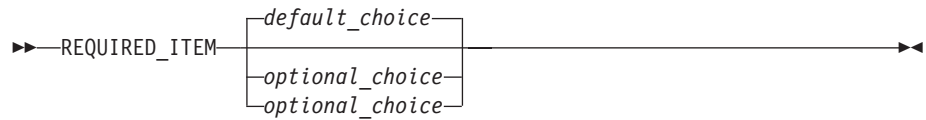
Fragments

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.

```
►►──STATEMENT──item 1──item 2──| A |──────────────────────────────────►◄
```

A:

```
    ┌─item 3─┐              ┌──────────┐
├───┼────────┼───KEYWORD────┼──────────┼───┤
    └─item 4─┘              ├─item 5─┤
                            └─item 6─┘
```
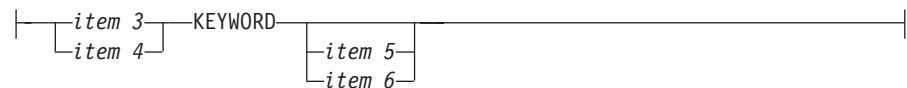
Substitution-block

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

```
►►──/VERB──────────LINE──line#──────────────────────────────────────►◄
         └─<A>─┘
```

where <A> is:

```
    ┌─EITHER─┐
►►──┼────────┼──────────────────────────────────────────────────────►◄
    └─OR─────┘
```
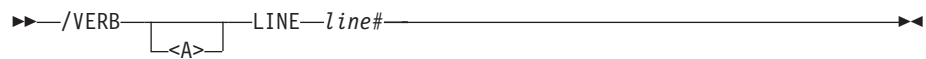
Parameter endings

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

```
►►──/MSVERIFY──────MSNAME──msname──────────────────────────────────►◄
              └─SYSID──sysid#─┘
```
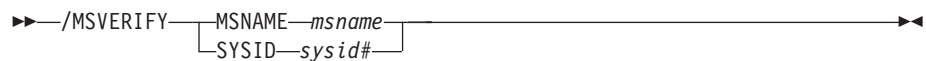
The MSNAME keyword in the example supports a name value and the SYSID keyword supports a number value.

How to read the notational symbols

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

- **Braces**, { }, indicate a choice of entry. Unless an item is underlined, indicating a default, or the items are enclosed in brackets, you must choose at least one of the entries.

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

Example of notation

The following example of PL/I syntax illustrates the notational symbols described In "How to read the notational symbols" on page xvii:

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

Interpret this example as follows:

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

Enhancements in this release

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

Debugging improvements

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

Performance improvements

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

Usability enhancements

- IEEE Decimal Floating-Point (DFP) is supported.

- The new MEMCONVERT built-in function will allow the user to convert arbitrary lengths of data between arbitrary code pages.
- The new ONOFFSET built-in function will allow the user to have easy access to another piece of information formerly available only in the runtime error message or dump, namely the offset in the user procedure at which a condition was raised.
- The new STACKADDR built-in function will return the address of the current dynamic save area (register 13 on z/OS) and will make it easier for users to write their own diagnostic code.
- The length of the mnemonic field in the assembler listing will be increased to allow for better support of the new z/OS instructions that have long mnemonics.
- More of the right margin will be used in the attributes, cross-reference and message listings.
- The CODEPAGE option will now accept 1026 (the Turkish code page) and 1155 (the 1026 code page plus the Euro symbol).
- The new MAXNEST option allows the user to flag excessive nesting of BEGIN, DO, IF and PROC statements.
- Under the new (and non-default) suboption NOELSEIF of the RULES option, the compiler will flag any ELSE statement that is immediately followed by an IF statement and suggest that it be rewritten as a SELECT statement.
- Under the new (and non-default) suboption NOLAXSTG of the RULES option, the compiler will flag declares where a variable A is declared as BASED on ADDR(B) and STG(A) > STG(B) not only (as the compiler did before) when B is AUTOMATIC, BASED or STATIC with constant extents but now also when B is a parameter declared with constant extents.
- The new QUOTE option will allow the user to specify alternate code points for the quote (″) symbol since this symbol is not code-page invariant.
- The new XML compiler option can be used to specify that the tags in the output of the XMLCHAR built-in function be either in all upper case or in the case in which they were declared.
- For compilations that produce no messages, the compiler will now include a line saying ″no compiler messages″ where the compiler messages would have been listed.
- The MACRO preprocessor will support a new suboption that will allow the user to specify whether it should process only %INCLUDE statements or whether it should process all macro statements.
- The integrated SQL preprocessor will now generate DB2 precompiler style declares for all *LOB_FILE, *LOCATOR, ROWID, BINARY and VARBINARY SQL types, in addition to the BLOB, CLOB and DBCLOB SQL types already supported, when the LOB(DB2) SQL preprocessor option is selected.

Enhancements from V3R6

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

DB2 V9 support

- Support for STDSQL(YES/NO)
- Support for CREATE TRIGGER (aka multiple SQL statements)
- Support for FETCH CONTINUE

- Support for SQL style comments ('--') embedded in SQL statements
- Support for additional SQL TYPES including
  - SQL TYPE IS BLOB_FILE
  - SQL TYPE IS CLOB_FILE
  - SQL TYPE IS DBCLOB_FILE
  - SQL TYPE IS XML AS
  - SQL TYPE IS BIGINT
  - SQL TYPE IS BINARY
  - SQL TYPE IS VARBINARY
- The SQL preprocessor will also now list the DB2 coprocessor options

Debugging improvements

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

Performance improvements

- Support, under ARCH(7), of the z/OS extended-immediate facility
- Exploitation of the CLCLU, MVCLU, PKA, TP and UNPKA instructions under ARCH(6)
- Exploitation of the CVBG and CVDG instructions under ARCH(5)
- Expanded use of CLCLE and MVCLE
- Conversions involving DB2 date-time patterns are now inlined
- The ALLOCATION built-in function is now inlined
- Conversions with a drifting $ have been inlined
- Conditional code has been eliminated from assignments to PIC'(n)Z'
- Conversions from FIXED BIN to a PICTURE that specifies a scale factor have been inlined
- Assignments to BIT variables that have inherited dimensions but which have strides that are divisible by 8 will now be inlined

Usability enhancements

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

- Flag assignments that can change a REFER object
- Flag use of KEYED DIRECT files without a KEY/KEYFROM clause
- Flag use of PICTURE as loop control variables

Enhancements from V3R5

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

Debugging improvements

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

Performance improvements

- Generation of branch-relative instructions so that the need for base registers and transfer vectors will be significantly eliminated
- Support, under ARCH(6), of the z/OS long displacement facility
- Simple structures using REFER will be mapped inlined rather than via a library call
- For structures using REFER still mapped via a library call, less code will be generated if the REFER specifies the bound for an array of substructures
- Faster processing of duplicate INCLUDEs
- Conversions to PICTURE variables with an I or R in the last position will now be inlined (such conversions had already been inlined when the last character was a T)
- Conversions to PICTURE variables ending with one or more B's will now be inlined if the corresponding picture without the B's would have been inlined
- Conversions to from CHARACTER to PICTURE variables consisting only of X's will now be inlined

Usability enhancements

- All parts of the listing, etc will count the source file as file 0, the first include file as file 1, the second (unique) include file as file 2, etc
- Conformance checking extended to include arrays
- Listings will include the build dates for any preprocessors invoked
- One-byte FIXED BINARY arguments can be suppressed for easier ILC with COBOL
- Alternate DD names may be specified for SYSADATA, SYSXMLSD and SYSDEBUG
- RULES(NOLAXMARGINS) tolerates, under XNUMERIC, sequence numbers
- RULES(NOUNREF) flags unreferenced AUTOMATIC variables
- If an assignment to a variable is done via library call, the message flagging the library call will include the name of the target variable
- Flag one-time DO loops
- Flag labels used as arguments

- Flag ALLOCATE and FREE of non-PARAMETER CONTROLLED in FETCHABLE if PRV used
- Flag DEFINED and BASED larger than their base even if the base is declared later
- Flag implicit FIXED DEC to 8-byte integer conversions

Enhancements from V3R4

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

Migration enhancements

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

Performance improvements

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

Usability enhancements

- Optionally provide offsets in the AGGREGATE listing in hex
- Support DEC(31) only when needed via the LIMITS(FIXEDDEC(15,31)) option

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

Debugging improvements

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

Enhancements from V3R3

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

More XML support

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

Improved performance

- The compilation time under OPT(2) will be significantly less than under Enterprise PL/I V3R2, especially for large programs.
- The compiler now uses the ED and EDMK instructions for inlined numeric conversions to PICTURE and CHARACTER. This results in faster, shorter code sequences and also in faster compilations.
- The compiler now generates better code for string comparisons. This also results in faster, shorter code sequences.
- The compiler now generates shorter, faster code for conversion from FIXED DECIMAL to PICTURE with trailing overpunch characters.

- The ARCH and TUNE compiler options now accept 5 as a valid sub-option. Under ARCH(5), the compiler will generate, when appropriate, some new z/Architecture instructions such as NILL, NILH, OILL, OILH, LLILL, and LLILH.

Easier migration

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

Improved usability

- The new LAXDEF suboption of the RULES compiler option allows the use of so-called illegal defining without having the compiler generate E-level messages.
- The new FLOATINMATH compiler option offers easier control of the precision with which math functions are evaluated.
- The new MEMINDEX, MEMSEARCH(R) and MEMVERIFY(R) built-in functions provide the ability to search strings larger than 32K.
- The new ROUTCDE and DESC suboptions of the DISPLAY(WTO) compiler option offers control of the corresponding elements of the WTO.
- The compiler now stores in each object a short string that will be in storage even when the associated code runs and that records all the options used to produce that object. This allows various tools to produce better diagnostics.
- The compiler now issues messages identifying more of the places where statements have been merged or deleted.
- The PLIDUMP output now includes a hex dump of user static.
- The PLIDUMP output now includes the options used to compile each program in the Language Environment traceback.
- The PLIDUMP output now includes more information on PL/I files.

Improved debug support

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

Enhancements from V3R2

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

Improved performance

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

Easier migration

- Compiler defaults have been changed for easier migration and compatibility. The changed defaults are:
  - CSECT
  - CMPAT(V2)
  - LIMITS(EXTNAME(7))
  - NORENT
- The compiler now honors the NOMAP, NOMAPIN and NOMAP attributes for PROCs and ENTRYs with OPTIONS(COBOL).
- The compiler now supports PROCs with ENTRY statements that have differing RETURNS attribute in the same manner as did the old host compiler.
- The compiler will now assume OPTIONS(RETCODE) for PROCs and ENTRYs with OPTIONS(COBOL).
- The SIZE condition is no longer promoted to ERROR if unhandled.
- Various changes have been made to reduce compile time and storage requirements.
- The OFFSET option will now produce a statement offset table much like the ones it produced under the older PL/I compilers.
- The FLAG option now has exactly the same meaning as it had under the old compilers, while the new MAXMSG option lets you decide if the compiler should terminate after a specified number of messages of a given severity. For example, with FLAG(I) MAXMSG(E,10), you can now ask to see all I-level messages while terminating the compilation after 10 E-level messages.
- The AGGREGATE listing now includes structures with adjustable extents.
- The STMT option is now supported for some sections of the listing.
- The maximum value allowed for LINESIZE has been changed to 32759 for F-format files and to 32751 for V-format files.

Improved usability

- The defaults for compiler options may now be changed at installation.
- The integrated SQL preprocessor now supports DB2 Unicode.

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

Enhancements from V3R1

This release also provides all of the functional enhancements offered in Enterprise PL/I V3R1, including the following:
- Support for multithreading on z/OS
- Support for IEEE floating-point on z/OS
- Support for the ANSWER statement in the macro prepreprocessor
- SAX-style XML parsing via the PLISAXA and PLISAXB built-in subroutines
- Additional built-in functions:
  - CS
  - CDS
  - ISMAIN
  - LOWERCASE
  - UPPERCASE

Enhancements from VisualAge PL/I

This release also provides all of the functional enhancements offered in VisualAge PL/I V2R2, including the following:
- Initial UTF-16 support via the WIDECHAR attribute

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

    If you create a WIDECHAR file, you should write the endianness flag ('fe_ff'wx) as the first two bytes of the file.
- DESCRIPTORS and VALUE options supported in DEFAULT statements
- PUT DATA enhancements

- POINTER, OFFSET and other non-computational variables supported
- Type-3 DO specifications allowed
- Subscripts allowed
- DEFINE statement enhancements
  - Unspecified structure definitions
  - CAST and RESPEC type functions
- Additional built-in functions:
  - CHARVAL
  - ISIGNED
  - IUNSIGNED
  - ONWCHAR
  - ONWSOURCE
  - WCHAR
  - WCHARVAL
  - WHIGH
  - WIDECHAR
  - WLOW
- Preprocessor enhancements
  - Support for arrays in preprocessor procedures
  - WHILE, UNTIL and LOOP keywords supported in %DO statements
  - %ITERATE statement supported
  - %LEAVE statement supported
  - %REPLACE statement supported
  - %SELECT statement supported
  - Additional built-in functions:
    - COLLATE
    - COMMENT
    - COMPILEDATE
    - COMPILETIME
    - COPY
    - COUNTER
    - DIMENSION
    - HBOUND
    - INDEX
    - LBOUND
    - LENGTH
    - MACCOL
    - MACLMAR
    - MACRMAR
    - MAX
    - MIN
    - PARMSET
    - QUOTE
    - REPEAT
    - SUBSTR
    - SYSPARM
    - SYSTEM
    - SYSVERSION
    - TRANSLATE
    - VERIFY

Part 1. Compiling your program

1

Chapter 1. Using compiler options and facilities

This chapter describes the options that you can use for the compiler, along with their abbreviations and IBM-supplied defaults. It's important to remember that PL/I requires access to Language Environment run time when you compile your applications. You can override most defaults when you compile your PL/I program. You can also override the defaults when you install the compiler.

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 3 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 3 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 79.

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

Compile-Time Option	Abbreviated Name	z/OS Default
AGGREGATE[(DEC|HEX)] | NOAGGREGATE	AG | NAG	NOAGGREGATE
ARCH(n)	–	ARCH(5)
ATTRIBUTES[(FULL|SHORT)] | NOATTRIBUTES	A | NA	NA [(FULL)][1]
BACKREG(5 | 11)	–	BACKREG(5)
BIFPREC(15 | 31)	–	BIFPREC(15)
BLANK('c')	–	BLANK('t')[2]
BLKOFF | NOBLKOFF	–	BLKOFF
CEESTART(FIRST | LAST)	–	CEESTART(FIRST)
CHECK(STORAGE | NOSTORAGE,CONFORMANCE | NOCONFORMANCE)	–	CHECK(NSTG, NOCONFORMANCE)
CMPAT(LE | V1 | V2)	–	CMPAT(V2)
CODEPAGE(n)	CP	CODEPAGE(1140)
COMMON | NOCOMMON	–	NOCOMMON
COMPACT | NOCOMPACT	–	NOCOMPACT

3

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

Compile-Time Option	Abbreviated Name	z/OS Default
COMPILE | NOCOMPILE[(W | E | S)]	C | NC	NOCOMPILE(S)
COPYRIGHT('string') | NOCOPYRIGHT	–	NOCOPYRIGHT
CSECT | NOCSECT	CSE | NOCSE	CSECT
CSECTCUT(n)	–	CSECTCUT(4)
CURRENCY('c')	CURR	CURRENCY($)
DBCS | NODBCS	–	NODBCS
DD(ddname-list)	–	DD(SYSPRINT,SYSIN, SYSLIB,SYSPUNCH, SYSLIN,SYSADATA, SYSXMLSD,SYSDEBUG)
DECIMAL(FOFLONASGN | NOFOFLONASGN, FORCEDSIGN, NOFORCEDSIGN)	DEC	DEC(FOFLONASGN, NOFORCEDSIGN)
DEFAULT(*attribute* | *option*)	DFT	See page 26
DISPLAY(STD | WTO(ROUTCDE(x) DESC(y) REPLY(z)))	–	DISPLAY(WTO)
DLLINIT | NODLLINIT	–	NODLLINIT
EXIT | NOEXIT	–	NOEXIT
EXTRN(FULL | SHORT)	–	EXTRN(FULL)
FLAG[(I | W | E | S)]	F	FLAG(W)
FLOAT(AFP(NOVOLATILE | VOLATILE) | NOAFP, DFP | NODFP)	–	FLOAT(AFP(NOVOLATILE) NODFP)
FLOATINMATH(ASIS | LONG | EXTENDED)	–	FLOATINMATH(ASIS)
GONUMBER | NOGONUMBER	GN | NGN	NOGONUMBER
GRAPHIC | NOGRAPHIC	GR | NGR	NOGRAPHIC
INCAFTER([PROCESS(*filename*)])	–	INCAFTER()
INCDIR('*directory name*') | NOINCDIR	–	NOINCDIR
INCPDS('*PDS name*') | NOINCPDS	–	NOINCPDS
INITAUTO | NOINITAUTO	–	NOINITAUTO
INITBASED | NOINITBASED	–	NOINITBASED
INITCTL | NOINITCTL	–	NOINITCTL
INITSTATIC | NOINITSTATIC	–	NOINITSTATIC
INSOURCE[(FULL|SHORT)] | NOINSOURCE	IS | NIS	NOINSOURCE
INTERRUPT | NOINTERRUPT	INT | NINT	NOINTERRUPT
LANGLVL(SAA | SAA2[,NOEXT | OS])	–	LANGLVL(SAA2,OS)
LIMITS(*options*)	–	See LIMITS on page 36
LINECOUNT(*n*)	LC	LINECOUNT(60)
LINEDIR | NOLINEDIR	–	NOLINEDIR
LIST | NOLIST	–	NOLIST
MACRO | NOMACRO	M | NM	NOMACRO
MAP | NOMAP	–	NOMAP
MARGINI('c') | NOMARGINI	MI | NMI	NOMARGINI
MARGINS(m,n[,c]) | NOMARGINS	MAR(m,n)	MARGINS F-format: (2,72) V-format: (10,100)
MAXMEM(n)	MAXM	MAXMEM(1048576)
MAXMSG(I | W | E | S,n)	–	MAXMSG(W,250)
MAXNEST(BLOCK(x) DO(y) IF(z))	–	MAXNEST(BLOCK(17) DO(17) IF(17))
MAXSTMT(n)	–	MAXSTMT(4096)
MAXTEMP(n)	–	MAXTEMP(50000)
MDECK | NOMDECK	MD | NMD	NOMDECK
NAME[('*external name*')] | NONAME	N	NONAME
NAMES('*lower*'[,*upper*])	–	NAMES('#@$','#@$')
NATLANG(ENU | UEN)	–	NATLANG(ENU)

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

Compile-Time Option	Abbreviated Name	z/OS Default
NEST | NONEST	–	NONEST
NOT	–	NOT('¬')
NUMBER | NONUMBER	NUM | NNUM	NUMBER
OBJECT | NOOBJECT	OBJ | NOBJ	OBJECT
OFFSET | NOOFFSET	OF | NOF	NOOFFSET
OPTIMIZE(0 | 2 | 3) | NOOPTIMIZE	OPT | NOPT	OPT(0)
OPTIONS[(ALL|DOC)] | NOOPTIONS	OP | NOP	NOOPTIONS
OR('c')	–	OR(' | ')
PP(*pp-name*) | NOPP	–	NOPP
PPCICS('string') | NOPPCICS	–	NOPPCICS
PPINCLUDE('string') | NOPPINCLUDE	–	NOPPINCLUDE
PPMACRO('string') | NOPPMACRO	–	NOPPMACRO
PPSQL('string') | NOPPSQL	–	NOPPSQL
PPTRACE | NOPPTRACE	–	NOPPTRACE
PREFIX(*condition*)	–	See page 50
PRECTYPE(ANS | DECRESULT)	–	PRECTYPE(ANS)
PROCEED | NOPROCEED[(W | E | S)]	PRO | NPRO	NOPROCEED(S)
PROCESS[(KEEP | DELETE)] | NOPROCESS	–	PROCESS(DELETE)
QUOTE('"')	–	QUOTE('"')
REDUCE | NOREDUCE	–	REDUCE
RENT | NORENT	–	NORENT
RESEXP | NORESEXP	–	RESEXP
RESPECT([DATE])	–	RESPECT()
RULES(*options*)	–	See RULES on page 54
SCHEDULER | NOSCHEDULER	–	NOSCHEDULER
SEMANTIC | NOSEMANTIC[(W | E | S)]	SEM | NSEM	NOSEMANTIC(S)
SERVICE('*service string*') | NOSERVICE	SERV | NOSERV	NOSERVICE
SOURCE | NOSOURCE	S | NS	NOSOURCE
SPILL(n)	SP	SPILL(512)
STATIC(FULL | SHORT)	–	STATIC(SHORT)
STDSYS | NOSTDSYS	–	NOSTDSYS
STMT | NOSTMT	–	NOSTMT
STORAGE | NOSTORAGE	STG | NSTG	NOSTORAGE
STRINGOFGRAPHIC(CHAR | GRAPHIC)	–	STRINGOFGRAPHIC (GRAPHIC)
SYNTAX | NOSYNTAX[(W | E | S)]	SYN | NSYN	NOSYNTAX(S)
SYSPARM('string')	–	SYSPARM('')
SYSTEM(MVS | CICS | IMS | TSO | OS)	–	SYSTEM(MVS)
TERMINAL | NOTERMINAL	TERM | NTERM	
TEST(*options*) | NOTEST	–	See "TEST" on page 65[3]
TUNE(n)	–	TUNE(5)
USAGE(*options*)	–	See USAGE on page 69
WIDECHAR(BIGENDIAN | LITTLEENDIAN)	WCHAR	WIDECHAR(BIGENDIAN)
WINDOW(w)	–	WINDOW(1950)
WRITABLE | NOWRITABLE[(FWS|PRV)]	–	WRITABLE
XINFO(*options*)	–	XINFO(NODEF,NOMSG, NOSYMNOSYN,NOXMI, NOXML)
XML(CASE(UPPER | ASIS))	–	XML(CASE(UPPER))
XREF[(FULL | SHORT)] | NOXREF	X | NX	NX [(FULL)][1]

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

Compile-Time Option	Abbreviated Name	z/OS Default

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.

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.

```
        ┌─NOAGGREGATE─────────────────────────┐
►►──────┴─AGGREGATE───────────────────────────┴──────────────────────►◄
                    │      ┌─DECIMAL─┐      │
                    └──(───┴─HEXADEC─┴───)──┘
```
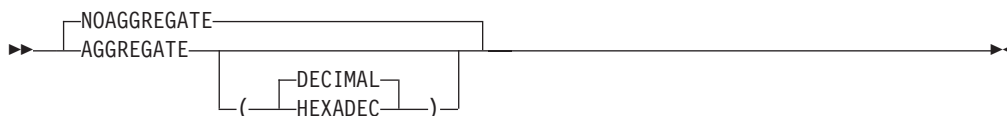
ABBREVIATIONS: AG, NAG

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

DECIMAL
All offsets are displayed in decimal.

HEXADEC
All offsets are displayed in hexadecimal.

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

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

ARCH

The ARCH option specifies the architecture for which the executable program's instructions are to be generated. It allows the optimizer to take advantage of specific hardware instruction sets. A subparameter specifies the group to which a model number belongs.

```
              ┌─5─┐
►►──ARCH──(───┴─n─┴───)───────────────────────────────────────────────►◄
```

The current values that may be specified for the ARCH level are:

0   Produces code that is executable on all models.

1   Produces code that executes only on the following and follow-on models:

9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900

9021-xx1 and 9021-xx2

9672-Rx1, 9672-Rx2(G1), 9672-Exx, and 9672-Pxx

2   Produces code that executes only on the following and follow-on models:

9672-Rx3(G2), 9672-Rx4(G3), 9672-Rx5(G4), and 2003

Specifically, these ARCH(2) machines and their follow-ons include the Branch Relative instruction set (Branch Relative and Save - BRAS), and the Halfword Immediate instruction set (for example, Add Halfword Immediate - AHI).

3   Produces code that executes only on the following and follow-on models:

9672-xx6 (G5) and 9672-xx7 (G6)

Specifically, these ARCH(3) machines and their follow-ons include a set of facilities for IEEE floating-point representation, as well as 12 additional floating-point registers and some new floating-point support instructions.

ARCH(3) (or higher) is required for execution of a program that uses IEEE floating-point.

4   Produces code that uses instructions available on model 2064-100 (z/900) in ESA/390 mode..

Specifically, these ARCH(4) machines support the LALR, BRCl and BRASL instructions, which are useful in generating relative address instructions, as well as the ALCR and SLBR instructions, which are useful in 8-byte integer arithmetic.

5   Produces code that uses instructions available on model 2064-100 (z/900) in z/Architecture mode.

Specifically, these ARCH(5) machines and their follow-ons include instructions such as NILL, NILH, OILL, OILH, LLILL and LLILH.

6   Produces code that uses instructions available on the 2084-xxx (z990) and 2086-xxx (z890) models in z/Architecture mode.

Specifically, the compiler on these ARCH(6) machines and their follow-ons may exploit the long-displacement instruction set. The long-displacement facility provides a 20-bit signed displacement field in 69 previously existing instructions (by using a previously unused byte in the instructions) and 44 new instructions. A 20-bit signed displacement allows relative addressing of up to 524,287 bytes beyond the location designated by a base register or base and index register pair and up to 524,288 bytes before that location. The enhanced previously existing instructions generally are ones that handle 64-bit binary integers. The new instructions generally are new versions of instructions for 32-bit binary integers. The new instructions also include

- A LOAD BYTE instruction that sign-extends a byte from storage to form a 32-bit or 64-bit result in a general register
- New floating-point LOAD and STORE instructions

The long-displacement facility provides register-constraint relief by reducing the need for base registers, code size reduction by allowing fewer instructions to be used, and additional improved performance through removal of possible address-generation interlocks.

7   Produces code that uses instructions available on the 2094-xxx models in z/Architecture mode.

Specifically, these ARCH(7) machines and their follow-ons add instructions supported by facilities such as extended-immediate and extended

translation facilities, which may be exploited by the compiler. For further information on these facilities, refer to z/Architecture Principles of Operation.

ARCH(7) machines also support the DFP instructions.

If you specify an ARCH value less than 5, the compiler will reset it to 5.

Note: The ″x″ in the model numbers above (such as 9672-Rx4 is a ″wildcard″ and stands for any alphanumeric machine of that type, such as 9627-RA4).

Note: Code that is compiled at ARCH(n) runs on machines in the ARCH(m) group if and only if m >= n.

ATTRIBUTES

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

```
        ┌─NOATTRIBUTES─┐
►►──────┼─ATTRIBUTES───┼──────────────────────────────────────►◄
                       │   ┌─FULL──┐   │
                       └─(─┼─SHORT─┼─)─┘
```
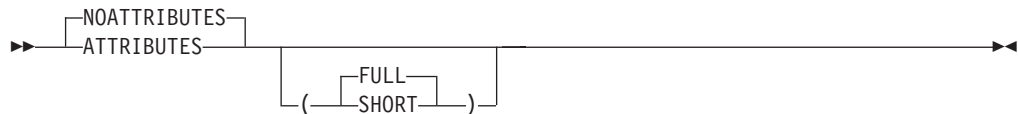
ABBREVIATIONS: A, NA, F, S

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

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

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

BACKREG

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

```
                   ┌─5──┐
►►──BACKREG──(─────┼─11─┼─)─────────────────────────────────────►◄
```

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

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

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

BIFPREC

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

```
              ┌─15─┐
►►──BIFPREC──(──┴─31─┴──)─────────────────────────────────────────►◄
```

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.

```
►►──BLANK──(─'──┬──char──┐──'─)──────────────────────────────►◄
                └────────┘
```

Note: Do not code any blanks between the quotes.

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

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.

BLKOFF

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

```
       ┌─BLKOFF───┐
►►──┴─NOBLKOFF─┴──────────────────────────────────────────────►◄
```

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

CEESTART

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

```
                    ┌─FIRST─┐
►►──CEESTART──(──┴─LAST──┴──)───────────────────────────────────►◄
```

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

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

Those users who want to use linker CHANGE cards must use the CEESTART(LAST) option.

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

CHECK

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

```
>>--CHECK--(--+-----------+--)-----------------------------><
              | ,         |
              v---------------+
              +-NOCONFORMANCE-+
              +-CONFORMANCE---+
              +-NOSTORAGE-----+
              +-STORAGE-------+
```
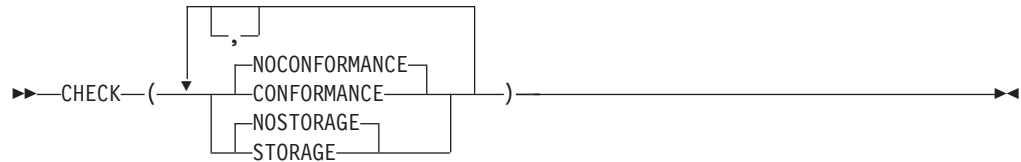
ABBREVIATIONS: STG, NSTG

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

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

This extra code will not be generated if the NODESCRIPTOR option applies to the procedure or if the block contains ENTRY statements or if the CMPAT(LE) option is in effect.

When you specify CHECK(STORAGE), the compiler calls slightly different library routines for ALLOCATE and FREE statements (except when these statements occur within an AREA). The following built-in functions, described in the PL/I Language Reference, can be used only when CHECK(STORAGE) has been specified:
- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

CMPAT

The CMPAT option specifies whether object compatibility with OS PL/I Version 1, OS PL/I Version 2, PL/I for MVS and VM, VisualAge PL/I for OS/390 or Enterprise PL/I for z/OS is to be maintained for programs sharing strings, AREAs, arrays and/or structures.

```
►►──CMPAT──(──┬──V2──┬──)────────────────────────────────────────────────►◄
              ├──LE──┤
              └──V1──┘
```

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

V1

Under CMPAT(V1), you can share strings, AREAs, arrays and/or structures with programs compiled with the OS PL/I compiler and with programs compiled with later PL/I compilers as long as the CMPAT(V1) option was used.

V2

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

DB2 stored procedures must be compiled with CMPAT(V1) or CMPAT(V2).

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

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

The DFT(DESCLIST) option conflicts with the CMPAT(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

```
►►──CODEPAGE──(──ccsid──)────────────────────────────────────────────────►◄
```

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
01025	01155		

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

COMMON

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

```
             ┌─NOCOMMON─┐
►►─────┬─────────────┬──────────────────────────────────────────────►◄
       └─COMMON──────┘
```

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

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

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

COMPACT

During optimizations performed during code generation, choices must be made between those optimizations which tend to result in faster but larger code and those which tend to result in smaller but slower code. The COMPACT option influences these choices. When the COMPACT option is used, the compiler favors those optimizations which tend to limit the growth of the code. Because of the interaction between various optimizations, including inlining, code compiled with the COMPACT option may not always generate smaller code and data.

```
             ┌─NOCOMPACT─┐
►►─────┬──────────────┬────────────────────────────────────────────►◄
       └─COMPACT──────┘
```

To evaluate the use of the COMPACT option for your application:
* Compare the size of the objects generated with COMPACT and NOCOMPACT
* Compare the size of the modules generated with COMPACT and NOCOMPACT
* Compare the execution time of a representative workload with COMPACT and NOCOMPACT

If the objects and modules are smaller with an acceptable change in execution time, then you can consider using COMPACT.

As new optimizations are added to the compiler, the behavior of the COMPACT option may change. You should re-evaluate the use of this option for each new release of the compiler and when the user changes the application code.

COMPILE

The COMPILE option causes the compiler to stop compiling after all semantic checking of the source program if it produces a message of a specified severity during preprocessing or semantic checking. Whether the compiler continues or not

depends on the severity of the error detected, as specified by the NOCOMPILE option in the list below. The NOCOMPILE option specifies that processing stops unconditionally after semantic checking.

```
        ┌─NOCOMPILE─────────────────────┐
        │         ┌─S─┐                 │
        │      ┌─(─┼─W─┼─)─┐            │
        │      │   └─E─┘   │            │
►►──────┴─COMPILE──────────┴────────────────────────►◄
```

ABBREVIATIONS: C, NC

COMPILE
> Generates code unless a severe error or unrecoverable error is detected. Equivalent to NOCOMPILE(S).

NOCOMPILE
> Compilation stops after semantic checking.

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

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

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

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

COPYRIGHT

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

```
     ┌─NOCOPYRIGHT──────────────────────────┐
►►───┴─COPYRIGHT──(──'copyright string'──)──┴─────────►◄
```
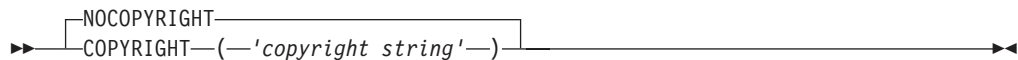
The string is limited to 1000 characters in length.

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

CSECT

The CSECT option ensures that the object module, if generated, contains named CSECTs. Use this option if you will be using SMP/E to service your product or to aid in debugging your program.

```
     ┌─CSECT───┐
►►───┴─NOCSECT─┴──────────────────────────────────────►◄
```

ABBREVIATIONS: CSE, NOCSE

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

Under the CSECT option, the code and static sections of your object module are given names that depend on the ″package name″ which is defined as follows:
- if the package statement was used, the ″package name″ is the leftmost label on the package statement
- otherwise, the ″package name″ is the leftmost label on the first procedure statement.

A ″modified package name″ of length 7 is then formed as follows:
- when the package name is less than 7 characters long, ″*″'s are prefixed to it to make a modified package name that is 7 characters long
- when the package name is more than 7 characters long, the first n and last $7 - n$ characters are used to make the modified package name, where the value n is set by the CSECTCUT option
- otherwise the package name is copied to the modified package name

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

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

So, for a package named ″SAMPLE″, the code csect name would be ″*SAMPLE1″, and the static csect name would be ″*SAMPLE2″.

CSECTCUT

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

```
►►──CSECTCUT──(──┬──4──┬──)──────────────────────────────────────►◄
                 └──n──┘
```

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

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

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

For example, for a compilation consisting of one procedure with the name BEISPIEL,
- under CSECTCUT(3), the compiler would collapse the name to BEIPIEL
- under CSECTCUT(4), the compiler would collapse the name to BEISIEL

CURRENCY

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

```
►►──CURRENCY──(──'──┬──$──┬──'──)──────────────────────────────────────►◄
                    └──x──┘
```

ABBREVIATIONS: CURR

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

DBCS

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

```
       ┌──NODBCS──┐
►►──────┼──────────┼──────────────────────────────────────────────────►◄
        └──DBCS────┘
```
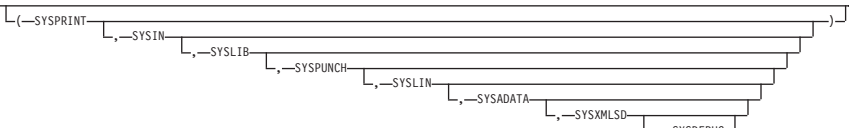
The NODBCS option will cause the listing, if generated, to show all DBCS
shift-codes as ″.″.

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

DD

The DD option allows you to specify alternate DD names for the various datasets
used by the compiler.

```
►►──DD──┬──────────────────────────────────────────────────────────────────┬──►◄
        └─(──SYSPRINT──┬───────────┬──────────────────────────────────────)─┘
                       └─,──SYSIN──┬───────────┬──────────────────────────┘
                                   └─,──SYSLIB──┬─────────────┬────────────┘
                                                └─,──SYSPUNCH──┬──────────┬──┘
                                                               └─,──SYSLIN─┬─────────────┬─┘
                                                                           └─,──SYSADATA──┬────────────┬─┘
                                                                                          └─,──SYSXMLSD──┬──────────────┬─┘
                                                                                                         └─,──SYSDEBUG──┘
```

Up to eight DD names may be specified. In order, they specify alternate DD names
for
- SYSPRINT
- SYSIN
- SYSLIB
- SYSPUNCH
- SYSLIN
- SYSADATA
- SYSXMLSD
- SYSDEBUG

If you wanted to use ALTIN as the DD name for the primary compiler source file,
you would have to specify DD(SYSPRINT,ALTIN). If you specified DD(ALTIN),
SYSIN would be used as the DDNAME for the primary compiler source file and
ALTIN would be used as the DD name for the compiler listing.

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

DECIMAL

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

```
>>--DECIMAL--(--+--------------------+--)--------------------><
                |  ,                 |
                v                    |
                +--FOFLONASGN-----+
                +--NOFOFLONASGN---+
                +--NOFORCEDSIGN---+
                +--FORCEDSIGN-----+
```
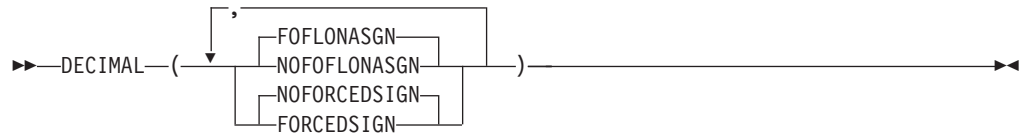
FOFLONASGN

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

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

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

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

FORCEDSIGN

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

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

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

DEFAULT

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

```
►►──DEFAULT──(─────────────────────────────────────────────)───────────►◄
             │        ┌─,─┐                                │
             │  ┌─────┴───┴─────┐                          │
             └──▼──┬─IBM─┬───────┘                         │
                   └─ANS─┘
                   ┌─EBCDIC─┐
                   ├────────┤
                   └─ASCII──┘
                   ┌─ASSIGNABLE────┐
                   ├───────────────┤
                   └─NONASSIGNABLE─┘
                   ┌─BYADDR──┐
                   ├─────────┤
                   └─BYVALUE─┘
                   ┌─NONCONNECTED─┐
                   ├──────────────┤
                   └─CONNECTED────┘
                   ┌─DESCRIPTOR───┐
                   ├──────────────┤
                   └─NODESCRIPTOR─┘
                   ┌─NATIVE────┐
                   ├───────────┤
                   └─NONNATIVE─┘
                   ┌─NATIVEADDR────┐
                   ├───────────────┤
                   └─NONNATIVEADDR─┘
                   ┌─NOINLINE─┐
                   ├──────────┤
                   └─INLINE───┘
                   ┌─ORDER───┐
                   ├─────────┤
                   └─REORDER─┘
                              ┌─OPTLINK──┐
                   ─LINKAGE──(─┴─────────┴─)─
                               └─SYSTEM─┘
                   ┌─EVENDEC───┐
                   ├───────────┤
                   └─NOEVENDEC─┘
                   ┌─NULL370─┐
                   ├─────────┤
                   └─NULLSYS─┘
                   ┌─BIN1ARG───┐
                   ├───────────┤
                   └─NOBIN1ARG─┘
                   ┌─NONRECURSIVE─┐
                   ├──────────────┤
                   └─RECURSIVE────┘
                   ┌─DESCLOCATOR─┐
                   ├─────────────┤
                   └─DESCLIST────┘
                              ┌─BYADDR──┐
                   ─RETURNS──(─┴─────────┴─)─
                               └─BYVALUE─┘
                   ┌─NOINITFILL──────────────────────┐
                   ├─INITFILL────────────────────────┤
                   └──────────(──init_value──)───────┘
                            ┌─HEXADEC─┐
                   ─SHORT──(─┴─────────┴─)─
                             └─IEEE────┘
                            ┌─ALIGNED───┐
                   ─DUMMY──(─┴───────────┴─)─
                             └─UNALIGNED─┘
                   ┌─UPPERINC─┐
                   ├──────────┤
                   └─LOWERINC─┘
                   ┌─NORETCODE─┐
                   ├───────────┤
                   └─RETCODE───┘
                   ┌─ALIGNED───┐
                   ├───────────┤
                   └─UNALIGNED─┘
                              ┌─MIN─┐
                   ─ORDINAL──(─┴─────┴─)─
                               └─MAX─┘
                   ┌─NOOVERLAP─┐
                   ├───────────┤
                   └─OVERLAP───┘
                   ┌─HEXADEC─┐
                   ├─────────┤
                   └─IEEE────┘
                          ┌─HEXADEC─┐
                   ─E──(──┴─────────┴──)─
                           └─IEEE────┘
```

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

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.

IBM is the default.

ASCII | EBCDIC

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

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

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

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

EBCDIC is the default.

ASSIGNABLE | NONASSIGNABLE

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

ASSIGNABLE is the default.

BYADDR | 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.

BYADDR is the default.

CONNECTED | 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.

NONCONNECTED is the default.

DESCRIPTOR | NODESCRIPTOR

Using DESCRIPTOR with a PROCEDURE indicates that a descriptor list was

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

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

DESCRIPTOR is the default.

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

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

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

Other combinations produce unpredictable results.

NATIVE is the default.

NATIVEADDR | 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.

NATIVEADDR is the default.

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

Specifying INLINE allows your code to run faster but, in some cases, also creates a larger executable file. For more information on how inlining can improve the performance of your application, see Chapter 11, "Improving performance," on page 269.

NOINLINE is the default.

ORDER | REORDER

Affects optimization of the source code. Specifying REORDER allows further optimization of your source code, see Chapter 11, "Improving performance," on page 269.

ORDER is the default.

LINKAGE

The linkage convention for procedure invocations is:

OPTLINK

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

SYSTEM

The standard linking convention for system APIs.

LINKAGE(OPTLINK) should be used for all routines called by or calling to JAVA, and it should also be used for all routines called by or calling to C (unless the C code has been compiled with a non-default linkage).

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

LINKAGE(OPTLINK) is the default.

EVENDEC | NOEVENDEC

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

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

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

EVENDEC is the default.

BIN1ARG | NOBIN1ARG

This suboption controls how the compiler handles one-byte REAL FIXED BIN arguments passed to an unprototyped function.

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

But under NOBIN1ARG, the compiler will assign any one-byte REAL FIXED BIN argument passed to an unprototyped function to a two-byte FIXED BIN temporary and pass that temporary instead.

Consider the following example:

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

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

If you specified DEFAULT(BIN1ARG), the compiler would pass the address of a one-byte FIXED BIN(1) argument to the routine f1 and the address of a two-byte FIXED BIN(15) argument to the routine f2. However, if you specified DEFAULT(NOBIN1ARG), the compiler would pass the address of a two-byte FIXED BIN(15) argument to both routines.

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

BIN1ARG is the default.

NULLSYS | NULL370

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

NULL370 is the default.

RECURSIVE | NONRECURSIVE

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

NONRECURSIVE is the default.

DESCLIST | DESCLOCATOR

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

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

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.

DESCLOCATOR is the default.

RETURNS (BYVALUE | BYADDR)

Sets the default for how values are returned by functions. See the *PL/I Language Reference* for more information.

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.

RETURNS(BYADDR) is the default.

INITFILL | NOINITFILL

This suboption controls the default initialization of automatic variables.

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

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

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

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

NOINITFILL is the default.

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

SHORT (HEXADEC) is the default.

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

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

Consider the following example:

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

dcl x entry( fixed bin(31) );

call x( b3 );
```

If you 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.

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

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

Under z/OS UNIX, the include name is built using the extension '.inc'. So, for example, under the DFT(LOWERINC) option, the statement %INCLUDE STANDARD; will cause the compiler to try to include *standard.inc*. But, under the DFT(UPPERINC) option, the statement %INCLUDE STANDARD; will cause the compiler to try to include *STANDARD.INC*.

UPPERINC is the default.

RETCODE | NORETCODE
If you specify RETCODE, for any external procedure that does not have the

RETURNS attribute, the compiler will generate extra code so that the procedure returns the integer value obtained by invoking the PLIRETV built-in function just prior to returning from that procedure.

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

NORETCODE is the default.

ALIGNED | 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.

ORDINAL(MIN | MAX)

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

ORDINAL(MIN) is the default.

OVERLAP | NOOVERLAP

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

NOOVERLAP will produce code that performs better; however, if you use NOOVERLAP, you must insure that the source and target never overlap.

NOOVERLAP is the default.

HEXADEC | IEEE

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

Programs that communicate with JAVA should probably use the IEEE option, and programs that pass data to or receive data from platforms where IEEE is the default representation for floating-point data might also want to use the IEEE option.

HEXADEC is the default.

E (HEXADEC | IEEE)

The E suboption determines how many digits will be used for the exponent in E-format items.

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

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

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

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

E(HEXADEC) is the default.

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

DISPLAY

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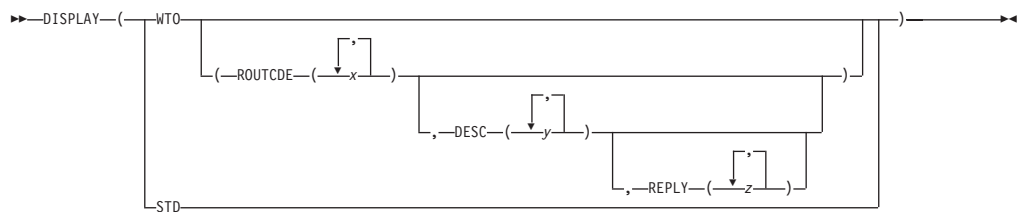
STD

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

WTO

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

The following suboptions are supported

ROUTCDE

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

DESC

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

REPLY

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

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

DLLINIT

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

```
      ┌─NODLLINIT─┐
►►─────┴─DLLINIT───┴──────────────────────────────────────────────────►◄
```

NODLLINIT has no effect on your programs.

EXIT

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

```
      ┌─NOEXIT───────────────────────────┐
►►─────┼─EXIT─────────────────────────────┼──────────────────────────►◄
                └─(───inparm_string───)─┘
```
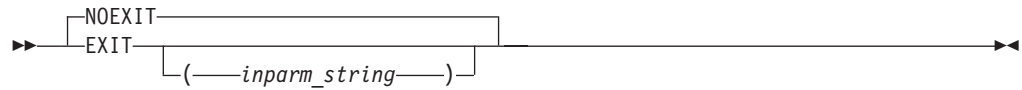
inparm_string
 A string that is passed to the compiler user exit routine during initialization.
 The string can be up to 31 characters long.

See Chapter 19, "Using user exits," on page 385 for more information on how to exploit this option.

EXTRN

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

```
                  ┌─FULL──┐
►►──EXTRN──(──┴─SHORT─┴──)────────────────────────────────────────►◄
```

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

SHORT
 EXTRNs are emitted only for those constants that are referenced.

FLAG

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

```
►►──FLAG─────────────────────────────────────────────────────────►◄
           │        ┌─W─┐        │
           └─(──┼─I─┼──)─┘
                    ├─E─┤
                    └─S─┘
```

ABBREVIATION: F

I List all messages.

W List all except information messages.

E List all except warning and information messages.

S List only severe error and unrecoverable error messages.

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

FLOAT

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

```
                    ┌──── , ◄────────────────────────────────┐
                    │         ┌─AFP─┐                         │
►►──FLOAT──(─────────▼─────────┴─────┴───────────────────────)──────────────►◄
                    │                   ┌─NOVOLATILE─┐        │
                    │              (────┴─VOLATILE───┴──)──   │
                    ├─NOAFP───────────────────────────────────┤
                    ├─NODFP─┐
                    └─DFP───┘
```
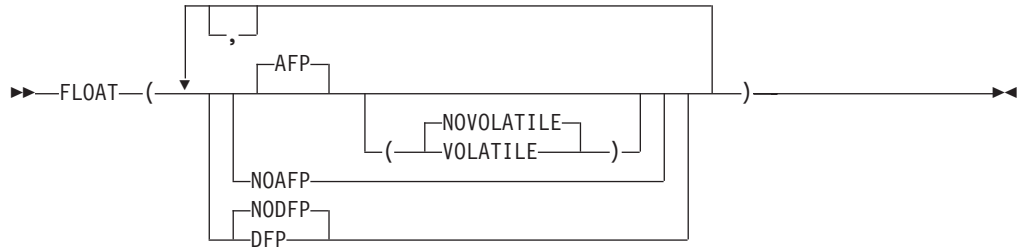
NOAFP
Compiler-generated code uses the traditional 4 floating-point registers.

AFP
Compiler-generated code may use all 16 floating-point registers.

> **VOLATILE**
> The compiler will not expect that the registers FPR8-FPR15 will be preserved by any called program. Consequently, the compiler will generate extra code to protect these registers.
>
> Extra code will be generated even if no floating-point registers are used, and this will hurt performance. If your application makes little or no use of floating-point, it would be much better to compile with FLOAT(NOAFP) than with FLOAT(AFP(VOLATILE)).

> **NOVOLATILE**
> The compiler will expect that the registers FPR8-FPR15 will be preserved by any called program (as z/OS says they should be). This option will produce the most optimal code and is highly recommended if you have no CICS code. However, it must be used with care in any code included in a CICS application. For a CICS application,
> - either: all code containing EXEC CICS statements must be compiled with FLOAT(AFP(VOLATILE))
> - or: all code using floating-point must be compiled with FLOAT(NOAFP) or with FLOAT(AFP(VOLATILE))
>
> It is safe and probably simplest not to use the FLOAT(AFP(NOVOLATILE)) option in any code that runs as part of a CICS application.

DFP
The DFP facility is exploited: all DECIMAL FLOAT data will be held in the DFP format described in the z/OS Principles of Operations manual and operations using DECIMAL FLOAT will be carried using the DFP hardware instructions described therein.

The ARCH option must be 7 (or greater) or this option will be rejected.

NODFP
The DFP facility is are not exploited.

Under FLOAT(DFP),
- the maximum precision for extended DECIMAL FLOAT will be 34 (not 33 as it is for hex float)

- the maximum precision for short DECIMAL FLOAT will be 7 (not 6 as it is for hex float)
- the values for DECIMAL FLOAT for the following built-ins will all have the appropriate changes
    - EPSILON
    - HUGE
    - MAXEXP
    - MINEXP
    - PLACES
    - RADIX
    - TINY
- the following built-in functions will all return the appropriate values for DECIMAL FLOAT (and values that will be much easier for a human to understand; for example SUCC(1D0) will be 1.000_000_000_000_001, and the ROUND function will round at a decimal place of course)
    - EXPONENT
    - PRED
    - ROUND
    - SCALE
    - SUCC
- decimal floating-point literals, when converted to ″right-units-view″, i.e. when the exponent has been adjusted, if needed, so that no non-zero digits follow the decimal point (for example, as would be done when viewing 3.1415E0 as 31415E-4), must have an exponent within the range of the normal numbers for the precision given by the literal. These bounds are given by the value of MINEXP-1 and MAXEXP-1. In particular, the following must hold
    - For short float, -95 <= exponent <= 90
    - For long float, -383 <= exponent <= 369
    - For extended float, -6143 <= exponent <= 6111
- when a DECIMAL FLOAT is converted to CHARACTER, the string will hold 4 digits for the exponent (as opposed to the 2 digits used for hexadecimal float)
- the IEEE and HEXADEC attributes will be accepted only if applied to FLOAT BIN, and the DEFAULT(IEEE/HEXADEC) option will apply only to FLOAT BIN.
- the mathematical built-in functions (ACOS, COS, SQRT, etc) will accept DECIMAL FLOAT arguments, but since the Language Environment support for these functions is incomplete, these functions will convert their arguments to IEEE BINARY FLOAT, invoke the corresponding IEEE BINARY FLOAT math routine and then convert the result back to DECIMAL FLOAT. For the same reason, DECIMAL FLOAT exponentiation will be handled in the analagous way.
- users of DFP need to be wary of the conversions that will arise in operations where one operand is FLOAT DECIMAL and the other is binary (i.e. FIXED BINARY, FLOAT BINARY or BIT). In such operations, the PL/I language rules dictate that the FLOAT DECIMAL operand be converted to FLOAT BINARY, and that conversion will require a library call. So, for example, for an assignment of the form $A = A + B;$ where A is FLOAT DECIMAL and B is FIXED BINARY, 3 conversions will be done and 2 of those will be library calls:
    1. A will be converted via library call from FLOAT DECIMAL to FLOAT BINARY
    2. B will be converted via inline code from FIXED BINARY to FLOAT BINARY

3. the sum *A + B* will be converted via library call from FLOAT BINARY to FLOAT DECIMAL

The use of the DECIMAL built-in function would help here: if the statement were changed to *A = A + DEC(B)*;, the library calls would be eliminated. The library calls could be eliminated by previously assign *B* to a FLOAT DECIMAL temporary variable and then adding that to *A*.

- the built-in function SQRTF will not be supported for DECIMAL FLOAT arguments (as there is no hardware instruction to which it can be mapped)
- DFP is not supported by the CAST type function.

FLOATINMATH

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

```
►►─FLOATINMATH─(──┬─ASIS─────┬──)─────────────────────────────►◄
                  ├─LONG─────┤
                  └─EXTENDED─┘
```

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.

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

GONUMBER

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

```
        ┌─NOGONUMBER─┐
►►──────┴─GONUMBER───┴──────────────────────────────────────────►◄
```

ABBREVIATIONS: GN, NGN

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

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

Note that there is no GOSTMT option. The only option that will produce information at run-time identifying where an error has occurred is the GONUMBER option. Also note that when the GONUMBER option is used, the term ″statement″ in the run-time error messages will refer to line numbers as used by the NUMBER compiler option - even if the STMT option was in effect.

GRAPHIC

The GRAPHIC option specifies that the source program can contain double-byte characters. The hexadecimal codes '0E' and '0F' are treated as the shift-out and shift-in control codes, respectively, wherever they appear in the source program, including occurrences in comments and string constants.

```
          ┌─NOGRAPHIC─┐
►►────────┴─GRAPHIC───┴──────────────────────────────────────────────►◄
```

ABBREVIATIONS: GR, NGR

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

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

INCAFTER

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

```
►►──INCAFTER──(─────────────────────────)───────────────────────────►◄
              └─PROCESS──(──filename──)─┘
```

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

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

Consider the following example:
```
INCAFTER(PROCESS(DFTS))
```

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

INCDIR

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

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

```
        ┌─NOINCDIR────────────────────┐
►►──────┴─INCDIR──(──'directory name'──)─┴──────────────────────────────►◄
```
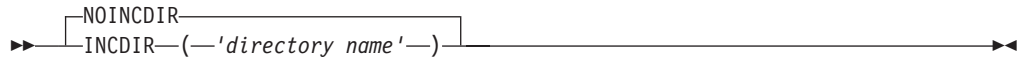
directory name
Name of the directory that should be searched for include files. You can specify the INCDIR option more than once and the directories are searched in order.

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

INCPDS

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

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

```
        ┌─NOINCPDS──────────────┐
►►──────┴─INCPDS──(──'PDS name'──)─┴──────────────────────────────────►◄
```

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

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

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

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

INITAUTO

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

```
        ┌─NOINITAUTO─┐
►►──────┴─INITAUTO───┴────────────────────────────────────────────────►◄
```

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

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

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

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

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

INITBASED

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

```
           ┌─NOINITBASED─┐
►►─────────┼─────────────┼──────────────────────────────────────►◄
           └─INITBASED───┘
```

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

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

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

INITCTL

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

```
           ┌─NOINITCTL─┐
►►─────────┼───────────┼──────────────────────────────────────────►◄
           └─INITCTL───┘
```

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

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

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

INITSTATIC

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

```
   ┌─NOINITSTATIC─┐
►►─┴─INITSTATIC───┴──────────────────────────────────────────►◄
```

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

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

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

INSOURCE

The INSOURCE option specifies that the compiler should include a listing of the source program before the PL/I macro preprocessor translates it.

```
   ┌─NOINSOURCE─────────────────────┐
►►─┴─INSOURCE──┬───────────────────┬─┴──────────────────────►◄
               │     ┌─FULL──┐     │
               └─(───┴─SHORT─┴───)─┘
```
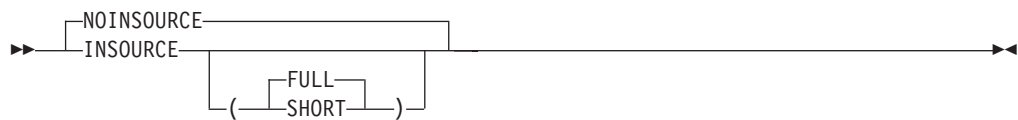
ABBREVIATION: IS, NIS

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

FULL is the default.

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

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).

INTERRUPT

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

```
  ┌─NOINTERRUPT─┐
►►─┤             ├────────────────────────────────────────────►◄
  └─INTERRUPT───┘
```

ABBREVIATION: INT, NINT

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

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

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

If you require the attention interrupt capability only for testing purposes, use the
TEST option instead of the INTERRUPT option. For more information see "TEST"
on page 65.

See Chapter 17, "Interrupts and attention processing," on page 379 for more
information about using interrupts in your programs.

LANGLVL

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

```
                    ┌─────,─────┐
                    │ ┌───────┐ │
►►─LANGLVL─(────────▼─┤ ┌─SAA2─┐ ├──)────────────────────────────►◄
                      ├─SAA───┤
                      ├─OS────┤
                      └─NOEXT─┘
```
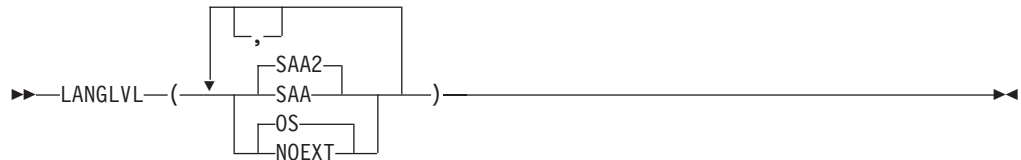
SAA

    The compiler flags keywords and other language constructs that are not
supported by OS PL/I Version 2 Release 3, and the compiler 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

    The only ENVIRONMENT options accepted are:

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

OS

All ENVIRONMENT options are allowed. For a complete list of the ENVIRONMENT options, see Table 13 on page 178.

LIMITS

The LIMITS option specifies various implementation limits.

```
>>--LIMITS--(--+--EXTNAME--(--+--7--+--)--+---------------------------------+--)--><
               |                  +--n--+                                    |
               |                                                             |
               +--FIXEDBIN--(--+--31--+--+-------------------+--)--+
               |                  +--63--+  +--,--+--31--+--+       |
               |                              +--63--+            |
               |                                                    |
               +--FIXEDDEC--(--+--15--+--+-------------------+--)--+
               |                  +--31--+  +--,--+--15--+--+       |
               |                              +--31--+            |
               |                                                    |
               +--NAME--(--+--100--+--)--------------------------+
                              +--n----+
```
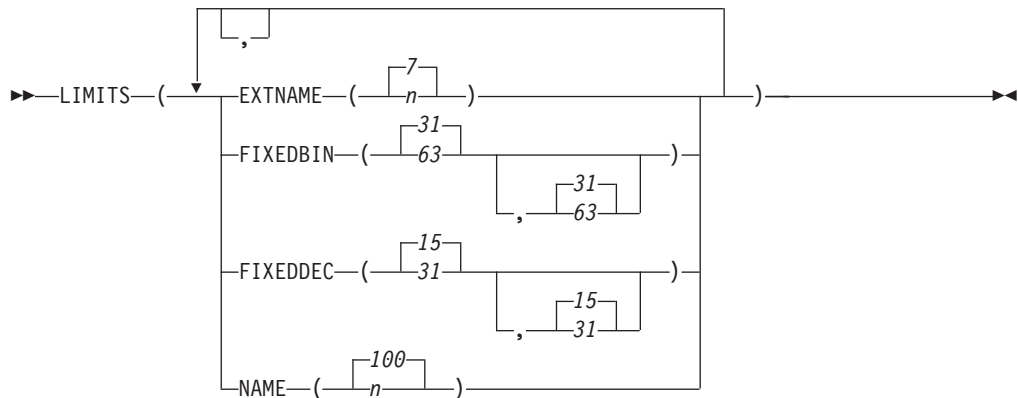
EXTNAME

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. The default is 15.

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

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

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

FIXEDDEC(31,15) is not allowed.

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 integer arithmetic will done using 4-byte integers.

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

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

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

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 31.

LINECOUNT

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

```
                        ┌─60─┐
►►──LINECOUNT──(──┴─n──┴──)────────────────────────────────────────────►◄
```

ABBREVIATION: LC

n   The number of lines in a page in the listing. The value can be from 10 to 32,767.

LINEDIR

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

```
     ┌─NOLINEDIR─┐
►►──┴─LINEDIR───┴──────────────────────────────────────────────────────►◄
```

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

LIST

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

```
     ┌─NOLIST─┐
►►──┴─LIST───┴─────────────────────────────────────────────────────────►◄
```

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

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

MACRO

The MACRO option invokes the MACRO preprocessor.

```
        ┌─NOMACRO─┐
►►──────┴─MACRO───┴──────────────────────────────────────────────►◄
```

ABBREVIATIONS: M, NM

You may also invoke the MACRO preprocessor via the PP(MACRO) option. For more discussion of the PP option, see "PP" on page 47.

For more discussion of the MACRO preprocessor, see "Macro preprocessor" on page 89.

MAP

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

```
        ┌─NOMAP─┐
►►──────┴─MAP───┴────────────────────────────────────────────────►◄
```

MARGINI

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

```
        ┌─NOMARGINI─┐
►►──────┴─MARGINI───┴──────(──'──c──'──)──────────────────────────►◄
```

ABBREVIATIONS: MI, NMI
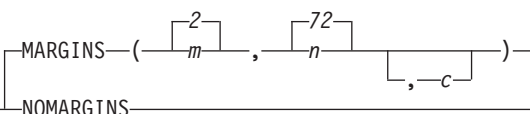
c   The character to be printed as the margin indicator.

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

MARGINS

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

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

```
        ┌─MARGINS──(──┬─2─┬──,──┬─72─┬───────────)─┐
        │             └─m─┘     └─n──┘   ┌──────┐  │
►►──────┤                                └─,──c─┘  ├──────────────►◄
        └─NOMARGINS────────────────────────────────┘
```

ABBREVIATION: MAR

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

n The column number of the rightmost character (last data byte) that is processed by the compiler. It should be greater than *m*, but must not exceed 200, except under MVS batch where it must not exceed 100.

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

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

(blank)
        Skip one line before printing

0   Skip two lines before printing

–   Skip three lines before printing

+   No skip before printing

1   Start new page

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

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

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

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

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

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

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

MAXMEM

When compiling with OPTIMIZE, the MAXMEM option limits the amount of memory used for local tables of specific, memory intensive optimizations to the

specified number of kilobytes. The minimum number of kilobytes that may be specified is 1. The maximum number of kilobytes that may be specified is 2097152, and the default is 1048576.

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

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

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

```
►►──MAXMEM──(size)──────────────────────────────────────────────►◄
```

ABBREVIATIONS: MAXM

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

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

MAXMSG

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

```
►►──MAXMSG─────────────────────────────────────────────────────►◄
              │  ┌──,──┐                    │
              │  │ ┌─W─┐ │                   │
              └─(─▼─┤ I ├──)─┘
                   ├─E─┤
                   └─S─┘
                   ┌─250─┐
                   └──n──┘
```
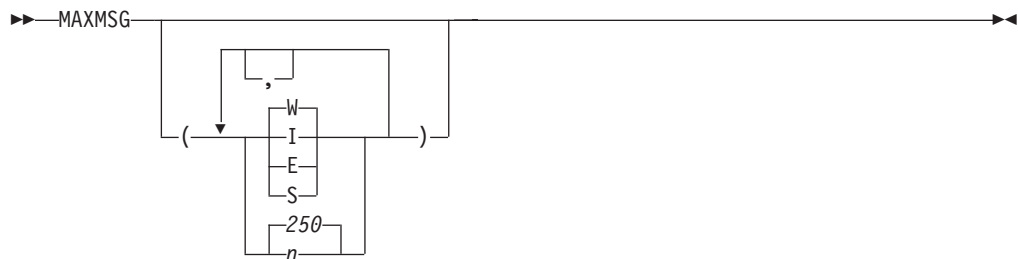
I Count all messages.

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.

MAXNEST

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

```
                        ┌──,──┐
                        │     │
►►──MAXNEST──(──▼──BLOCK──(──┬──17──┬──)──┬──)──────────────►◄
              │            └──x───┘       │
              ├──DO──(──┬──17──┬──)───────┤
              │         └──x───┘          │
              └──IF──(──┬──17──┬──)────────┘
                        └──z───┘
```
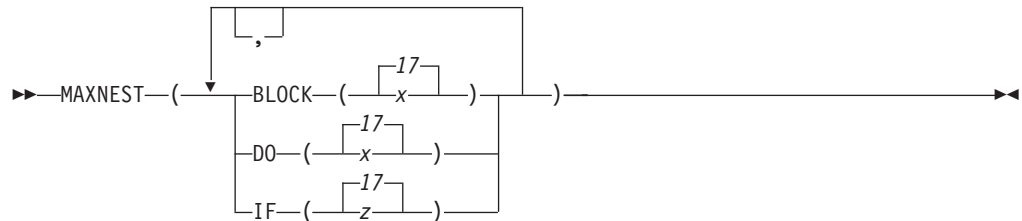
BLOCK
> Specifies the maximum nesting of BEGIN and PROCEDURE statements.

DO
> Specifies the maximum nesting of DO statements.

IF Specifies the maximum nesting of IF statements.

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

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

MAXSTMT

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

```
►►──MAXSTMT──(size)──────────────────────────────────────►◄
```

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

The default for MAXSTMT is 4096.

MAXTEMP

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

```
►►──MAXTEMP──(──max──)────────────────────────────────────►◄
```

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

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

```
          ┌─NOMDECK─┐
►►─────────┼─────────┼──────────────────────────────────────────────────►◄
           └─MDECK───┘
```

ABBREVIATIONS: MD, NMD

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

NAME

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

```
          ┌─NONAME──────────┐
►►────────┼─────────────────┼─────────────────────────────────────────────►◄
          └─NAME──┬───────────┬─┘
                  └─(─'name'─)─┘
```

ABBREVIATIONS: N

If no 'name' is specified as a suboption of the NAME option, then the 'name' used is determined as follows
- if there is a PACKAGE statement, the leftmost name on it is used
- otherwise, the leftmost name on the first PROCEDURE statement is used

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

NAMES

The NAMES option specifies the *extralingual characters* that are allowed in identifiers. Extralingual characters are those characters other than the 26 alphabetic, 10 digit, and special characters defined in *PL/I Language Reference*.

```
                   ┌──────────────────┐
►►──NAMES──(─'─────▼─extraling_char──┬─'──┬──────────────────────────────┬──)──►◄
                                            │   ┌──────────────────────┐  │
                                            └─┬─'─▼─upp_extraling_char─┬─'─┘
                                              └─,─┘
```
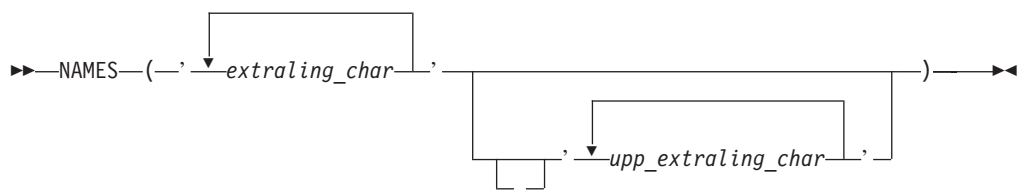
extralingual_char
An extralingual character

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

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

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

NATLANG

The NATLANG option specifies the ″language″ for compiler messages, headers, etc.

```
                    ┌─ENU─┐
►►─NATLANG─(─┤     ├─)──────────────────────────────────────◄◄
                    └─UEN─┘
```

ENU
All compiler messages, headers etc will be in mixedcase English.

UEN
All compiler messages, headers etc will be in uppercase English.

NEST

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

```
        ┌─NONEST─┐
►►─┤          ├──────────────────────────────────────────────◄◄
        └─NEST───┘
```

NOT

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

```
            ┌──────────┐
►►─NOT─('─▼─char─┘─'─)─────────────────────────────────────────◄◄
```

char
A single SBCS character.

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

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

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

recognized. Similarly, NOT('~¬') means that either the tilde or the standard NOT symbol will be recognized as the logical NOT operator.

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

NUMBER

The number option specifies that statements in the source program are to be identified by the line and file number of the file from which they derived and that this pair of numbers is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, LIST, MAP, OFFSET, SOURCE and XREF options. The File Reference Table at the end of the listing shows the number assigned to each of the input files read during the compilation.

```
          ┌─NUMBER───┐
►►────────┴─NONUMBER─┴──────────────────────────────────────────◄◄
```

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.

Also note that in the pseudo-assembler listing produced by the LIST option, the file number is left blank for the first file.

The default is NUMBER.

OBJECT

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

```
          ┌─OBJECT───┐
►►────────┴─NOOBJECT─┴──────────────────────────────────────────◄◄
```

ABBREVIATIONS: OBJ, NOBJ

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

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

OFFSET

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

```
   ┌─NOOFFSET─┐
►►─┴─OFFSET───┴──────────────────────────────────────────────────►◄
```

OPTIMIZE

The OPTIMIZE option specifies the type of optimization required:

```
   ┌─NOOPTIMIZE────────────────────┐
►►─┼─OPTIMIZE──────────────────────┼──────────────────────────────►◄
                │   ┌─2─┐         │
                └─(─┼─TIME─┼─)─────┘
                    ├─0─┤
                    └─3─┘
```
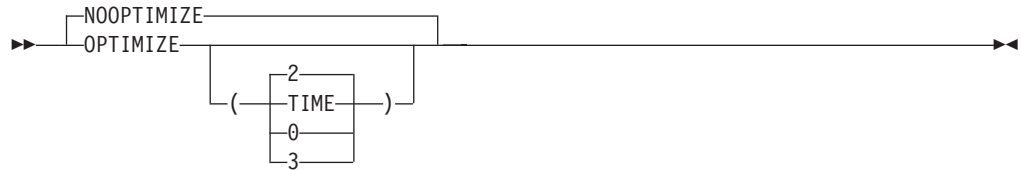
ABBREVIATIONS: OPT, NOPT

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

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

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

It is strongly recommended that the DFT(REORDER) option be used with the OPTIMIZE option. In fact, the effect of OPTIMIZE is severely limited for any PROCEDURE or BEGIN-block for which all of the following are true:
- The ORDER option applies to the block.
- The block contains ON-units for hardware-detected conditions (such as ZERODIVIDE).
- The block has labels that are the (potential) target of branches out of those ON-units.

The use of OPTIMIZE(2) could result in a substantial increase in compile time over NOOPTIMIZE and a substantial increase in the space required. For example, compiling a large program at OPTIMIZE(2) might take several minutes and could require a region of 75M or more.

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

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

NOOPTIMIZE is the equivalent of OPTIMIZE(0).

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

Note that the use of OPTIMIZE(2) or OPTIMIZE(3) severely limits the functionality of the TEST option. For example:
- if the HOOK suboption of TEST is in effect, then only block hooks will be generated
- if the NOHOOK suboption of TEST is in effect, attempts to list or change a variable may fail (because the variable may have been optimized into a register) and attempts to stop at a particular statement may cause the debugger to stop several times (because the statement may have split up into several parts)

For more information on choosing the best options to improve the performance of your code, see Chapter 11, "Improving performance," on page 269.

OPTIONS

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

```
           ┌─NOOPTIONS──────────────┐
►►─────────┴─OPTIONS────────────────┴──────────────────────────────►◄
                    │        ┌─DOC─┐ │
                    └─( ─────┴─ALL─┴─)─┘
```
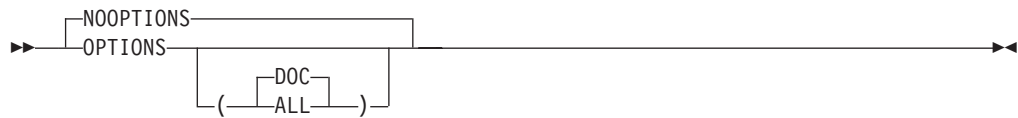
ABBREVIATIONS: OP, NOP

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

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

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

OR

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

```
        ┌───────────────────┐
►►──OR──▼──( ─' ─char─' ─)──┴──────────────────────────────────────►◄
```

Note: Do not code any blanks between the quotes.

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

char
> A single SBCS character.

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

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

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

PP

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

```
          ┌─NOPP──────────────────────────────────────┐
          │            ┌────────────────┐             │
          │            │      ┌─,─┐      │             │
►►────────┼─PP──(──────▼──pp-name────────┴──────)──────┴──────────────►◄
                                 └─(──pp-string──)─┘
```
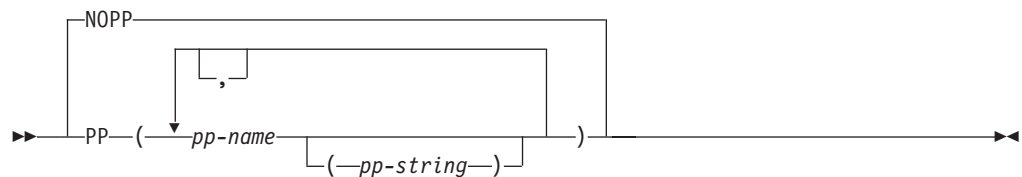
pp-name
> The name given to a particular preprocessor. CICS, INCLUDE, MACRO and SQL are the only preprocessors currently supported. Using an undefined name causes a diagnostic error.

pp-string
> A string, delimited by quotes, of up to 100 characters representing the options for the corresponding preprocessor. For example, PP(MACRO('CASE(ASIS)')) invokes the MACRO preprocessor with the option CASE(ASIS).

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

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

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

If you specify the PP option more than once, the compiler effectively concatenates them. So specifying PP(SQL) PP(CICS) is the same as specifying PP(SQL CICS). This also means that if you specified PP(MACRO SQL('OPTIONS')) and PP(MACRO SQL('OPTIONS DATE(ISO)')), then the resulting PP option would be PP(MACRO SQL('OPTIONS') MACRO SQL('OPTIONS DATE(ISO)')) and both the MACRO and SQL preprocessor would be invoked twice. If you were doing this in an attempt to override the earlier SQL options, it might be better not to specify the preprocessor options in the PP option, but rather to specify them via the PPSQL option, i.e. specify PP(MACRO SQL) PPSQL('OPTIONS DATE(ISO)').

For more discussion of the preprocessors, see Chapter 2, "PL/I preprocessors," on page 87.

PPCICS

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

```
   ┌─NOPPCICS─────────────────────────┐
►►─┴─PPCICS──(──'options string'──)─┴──────────────────────────►◄
```

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

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

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

PPINCLUDE

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

```
   ┌─NOPPINCLUDE──────────────────────────┐
►►─┴─PPINCLUDE──(──'options string'──)─┴──────────────────────►◄
```

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

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

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

PPMACRO

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

```
   ┌─NOPPMACRO────────────────────────────┐
►►─┴─PPMACRO──(──'options string'──)─┴────────────────────────►◄
```

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

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

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

PPSQL

The PPSQL option specifies options to be passed to the SQL preprocessor if it is invoked.

```
       ┌─NOPPSQL──────────────────────┐
►►─────┴─PPSQL──(──'options string'──)─┴──────────────────────────────►◄
```

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

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

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

PPTRACE

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

```
       ┌─NOPPTRACE─┐
►►─────┴─PPTRACE───┴───────────────────────────────────────────────────►◄
```

PRECTYPE

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

```
              ┌─ANS──────┐
►►──PRECTYPE──(──┼─DECDIGIT──┼──)────────────────────────────────────────────►◄
              └─DECRESULT─┘
```

ANS

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

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

DECDIGIT

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

DECRESULT

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

PREFIX

The PREFIX option enables or disables the specified PL∕I conditions in the compilation unit being compiled without your having to change the source program. The specified condition prefixes are logically prefixed to the beginning of the first PACKAGE or PROCEDURE statement.

```
►►──PREFIX──(─────────────────────)────────────────────────────────────────►◄
            │         ┌─,─┐        │
            │         ▼    │       │
            └──────────condition───┘
```
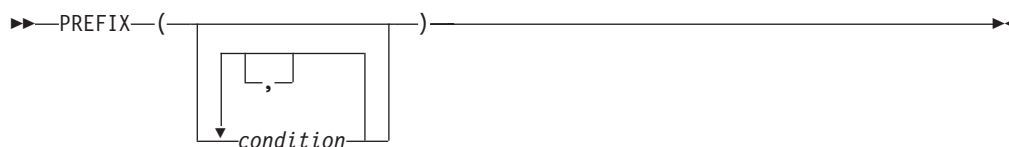
condition

Any condition that can be enabled∕disabled in a PL∕I program, as explained in *PL∕I Language Reference.*

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

PROCEED

The PROCEED option stops the compiler after processing by a preprocessor is completed depending on the severity of messages issued by previous preprocessors.

```
          ┌─NOPROCEED─┐ ┌──────────────┐
          │           │ │       ┌─S─┐  │
          │           │ └─(──┬──┼─W─┤──┬─)─┘
          │           │      │   └─E─┘  │
►►────────┴─PROCEED───┴──────┴──────────┴────────────────────────►◄
```

ABBREVIATIONS: PRO, NPRO

PROCEED
> Is equivalent to NOPROCEED(S).

NOPROCEED
> Ends the processing after the preprocessor has finished compiling.

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

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

NOPROCEED(W)
> The invocation of preprocessors and the compiler does not continue if a warning, error, severe error, or unrecoverable error is detected in this stage of preprocessing.

PROCESS

The PROCESS option determines if *PROCESS statements are allowed and, if they are allowed, if they are written to the MDECK file.

```
        ┌─PROCESS─┐ ┌──────────────────┐
        │         │ │       ┌─DELETE─┐  │
        │         │ └─(──┬──┼─KEEP───┤──┬─)─┘
        │         │      │            │
►►──────┴─────────┴──────┴────────────┴──────────────────────────►◄
        └─NOPROCESS─────────────────────────────┘
```
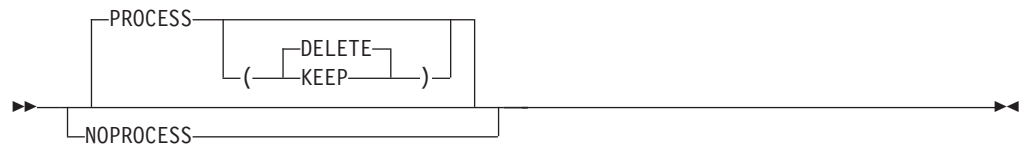
Under the NOPROCESS option, the compiler will flag any *PROCESS statement with an E-level message.

Under the PROCESS(KEEP) option, the compiler will not flag *PROCESS statements, and the compiler will retain any *PROCESS statements in the MDECK output.

Under the PROCESS(DELETE) option, the compiler will not flag *PROCESS statements, but the compiler will not retain any *PROCESS statements in the MDECK output.

QUOTE

The QUOTE option specifies up to seven alternate symbols that can be used as the quote character.

```
                    ┌─"─┐
►►──QUOTE──(──'──┬─char─┬──'──)──────────────────────────────────────►◄
```

Note: Do not code any blanks between the quotes.

The IBM-supplied default code point for the QUOTE symbol is '"'.

char
    A single SBCS character.

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

The QUOTE option is ignored if the GRAPHIC option is also specified.

REDUCE

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

The REDUCE option also allows the compiler to reduce the assignment of matching structures into a simple aggregate move - even if the structures contain POINTER fields.

```
     ┌─REDUCE───┐
►►───┴─NOREDUCE─┴──────────────────────────────────────────────────►◄
```

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 *field12* and *field13*.

```
 dcl
  1 sample ext,
    5  field10        bin fixed(31),
    5  field11        bin fixed(15),
    5  field12        bit(8),
    5  field13        bin fixed(31);
```

Now consider the assignment *sample = '';*

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

However, under REDUCE, the assignment would be reduced to one operation, but the padding byte will be zeroed out.

The NOREDUCE option makes the compiler act more like the OS PL/I and the PL/I for MVS compilers. These compilers would reduce an assignment of matching structures into a simple aggregate move unless the structures contain POINTER fields. The NOREDUCE option will make this compiler act the same way.

RENT

```
        ┌─NORENT─┐
►►──────┴─RENT───┴──────────────────────────────────────────────────◄◄
```
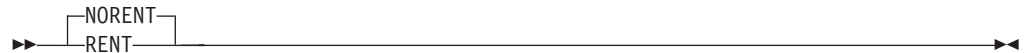
Your code is "naturally reentrant" if it does not alter any of its static variables.

The RENT option specifies that the compiler is to take code that is not naturally reentrant and make it reentrant. Refer to the z/OS Language Environment Programming Guide for a detailed description of reentrancy. If you use the RENT option, the Linkage Editor cannot directly process the object module that is produced: you must use either the prelinker or PDSE's.

The NORENT option specifies that the compiler is not to specifically generate reentrant code from non-reentrant code. Any naturally reentrant code remains reentrant.

If you link a module (either MAIN or FETCHABLE) containing one or more programs compiled with the RENT option, you must specify DYNAM=DLL and REUS=RENT on the link step.

If you specify the options NORENT and LIMITS(EXTNAME(n)) (with $n <= 7$), then the text decks generated by the compiler will have the same format as those generated by the older PL/I compilers. This means that the prelinker would not be needed to create a PDS-style load module. If you use any other options, you must use either the prelinker or PDSE's.

The code generated under the NORENT option may not be reentrant unless the NOWRITABLE option is also specified.

The use of the NORENT does preclude the use of some features of the compiler. In particular:
- DLLs cannot be built
- reentrant, writeable static is not supported
- a STATIC ENTRY VARIABLE cannot have an INITIAL value

You may mix RENT and NORENT code subject to the following restrictions:
- code compiled with RENT cannot be mixed with code compiled with NORENT if they share any EXTERNAL STATIC variables
- code compiled with RENT cannot call an ENTRY VARIABLE set in code compiled with NORENT
- code compiled with RENT cannot call an ENTRY CONSTANT that was fetched in code compiled with NORENT

- code compiled with RENT can fetch a module containing code compiled with NORENT if one of the following is true
  - all the code in the fetched module was compiled with NORENT
  - the code containing the entry point to the module was compiled with RENT
- code compiled with NORENT code cannot fetch a module containing any code compiled with RENT
- code compiled with NORENT WRITABLE cannot be mixed with code compiled with NORENT NOWRITABLE if they share any external CONTROLLED variables or any external FILEs

Given the above restrictions, the following is still valid:
- a NORENT routine, called say *mnorent*, statically links and calls a RENT routine, called say *mrent*
- the RENT routine *mrent* then fetches and CALLs a separately-linked module with an entry point compiled with RENT

RESEXP

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

```
        ┌─RESEXP──┐
►►──────┴─NORESEXP─┘──────────────────────────────────────────────►◄
```

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

The RESPECT option causes the compiler to honor any specification of the DATE attribute and to apply the DATE attribute to the result of the DATE built-in function.

```
►►──RESPECT──(──────────)─────────────────────────────────────────►◄
              └─DATE─┘
```
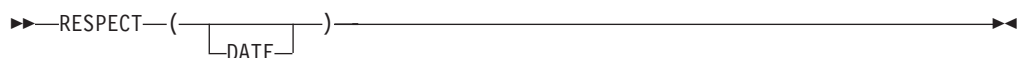
Using the default, RESPECT(), causes the compiler to ignore any specification of the DATE attribute and ensures that the compiler does not apply the DATE attribute to the result of the DATE built-in function.

RULES

The RULES option allows or disallows certain language capabilities and lets you choose semantics when alternatives are available. It can help you diagnose common programming errors.

```
         ┌──────────────,◄──────────────────┐
►►─RULES─(─▼─┬──IBM─────┬──────────────────────)──────────►◄
            ├──ANS──────┤
            ├──BYNAME────┤
            ├──NOBYNAME──┤
            ├──NODECSIZE─┤
            ├──DECSIZE───┤
            ├──ELSEIF────┤
            ├──NOELSEIF──┤
            ├──EVENDEC───┤
            ├──NOEVENDEC─┤
            ├──GOTO──────┤
            ├──NOGOTO────┤
            ├──NOLAXBIF──┤
            ├──LAXBIF────┤
            ├──NOLAXCTL──┤
            ├──LAXCTL────┤
            ├──LAXDCL────┤
            ├──NOLAXDCL──┤
            ├──NOLAXDEF──┤
            ├──LAXDEF────┤
            ├──LAXIF─────┤
            ├──NOLAXIF───┤
            ├──LAXINOUT──┤
            ├──NOLAXINOUT┤
            ├──LAXLINK───┤
            ├──NOLAXLINK─┤
            ├──LAXMARGINS─────────────────────┤
            ├──NOLAXMARGINS───────────────────┤
            │            ┌──STRICT───┐        │
            │       └──(──┴──XNUMERIC─┴──)──┘  │
            ├──LAXPUNC───┤
            ├──NOLAXPUNC─┤
            ├──LAXQUAL───┤
            ├──NOLAXQUAL─┤
            ├──LAXSEMI───┤
            ├──NOLAXSEMI─┤
            ├──LAXSTG────┤
            ├──NOLAXSTG──┤
            ├──NOLAXSTRZ─┤
            ├──LAXSTRZ───┤
            ├──MULTICLOSE──┤
            ├──NOMULTICLOSE┤
            ├──UNREF─────┤
            └──NOUNREF───┘
```

IBM | ANS

Under the IBM suboption:

- For operations requiring string data, data with the BINARY attribute is converted to BIT.
- Conversions in arithmetic operations or comparisons occur as described in the *PL/I Language Reference.*
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *PL/I Language Reference* except that operations specified as scaled fixed binary are evaluated as scaled fixed decimal.
- Nonzero scale factors are permitted in FIXED BIN declares.

- If the result of any precision-handling built-in function (ADD, BINARY, etc.) has FIXED BIN attributes, the specified or implied scale factor can be nonzero.
- Even if all arguments to the MAX or MIN built-in functions are UNSIGNED FIXED BIN, the result is always SIGNED.
- Even when you add, multiply, or divide two UNSIGNED FIXED BIN operands, the result has the SIGNED attribute.
- Even when you apply the MOD or REM built-in functions to two UNSIGNED FIXED BIN operands, the result has the SIGNED attribute.

Under the ANS suboption:
- For operations requiring string data, data with the BINARY attribute is converted to CHARACTER.
- Conversions in arithmetic operations or comparisons occur as described in the *PL/I Language Reference*.
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *PL/I Language Reference*.
- Nonzero scale factors are **not** permitted in FIXED BIN declares.
- If the result of any precision-handling built-in function (ADD, BINARY, etc.) has FIXED BIN attributes, the specified or implied scale factor must be zero.
- If all arguments to the MAX or MIN built-in functions are UNSIGNED FIXED BIN, the result is UNSIGNED.
- When you add, multiply or divide two UNSIGNED FIXED BIN operands, the result has the UNSIGNED attribute.
- When you apply the MOD or REM built-in functions to two UNSIGNED FIXED BIN operands, the result has the UNSIGNED attribute.

Also, under RULES(ANS), the following errors, which the old compilers ignored, will produce E-level messages
- Specifying a string constant as the argument to the STRING built-in
- Giving too many asterisks as subscripts in an array reference
- Qualifying a CONTROLLED variable with a POINTER reference (as if the CONTROLLED variable were BASED)

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

DECSIZE | NODECSIZE
Specifying DECSIZE causes the compiler to flag any assignment of a FIXED DECIMAL expression to a FIXED DECIMAL variable when the SIZE condition is disabled if the SIZE condition could be raised by the assignment.

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

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

ELSEIF | NOELSEIF
Specifying NOELSEIF causes the compiler to flag any ELSE statement that is immediately followed by an IF statement and suggest that it be rewritten as a SELECT statement.

This option can be useful in enforcing that SELECT statements be used rather than a series of nested IF-THEN-ELSE statements.

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

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

LAXBIF | NOLAXBIF
Specifying LAXBIF causes the compiler to build a contextual declaration for built-in functions, such as NULL, even when used without an empty parameter list.

LAXCTL | NOLAXCTL
Specifying LAXCTL allows a CONTROLLED variable to be declared with a constant extent and yet to be allocated with a differing extent. NOLAXCTL requires that if a CONTROLLED variable is to be allocated with a varying extent, then that extent must be specified as an asterisk or as a non-constant expression.

The following code is illegal under NOLAXCTL:

```
dcl a bit(8) ctl;
alloc a;
alloc a bit(16);
```

But this code would still be valid under NOLAXCTL:

```
dcl b bit(n) ctl;
dcl n fixed bin(31) init(8);
alloc b;
alloc b bit(16);
```

LAXDCL | 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 | NOLAXIF
Specifying RULES(NOLAXIF) will cause the compiler to flag any IF, WHILE, UNTIL, and WHEN clauses that do not have the attributes BIT(1) NONVARYING.

The following would all be flagged under NOLAXIF:

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

LAXINOUT | NOLAXINOUT
Specifying NOLAXINOUT causes the compiler to assume that all

ASSIGNABLE BYADDR parameters are input (and possibly output) parameters and hence to issue a warning if it thinks such a parameter has not been initialized.

LAXLINK | NOLAXLINK

Specifying NOLAXLINK causes the compiler to flag any assign or compare of two ENTRY variables or constants if any of the following do not match:

- the parameter description lists

  For instance if A1 is declared as ENTRY(CHAR(8)) and A2 as ENTRY(POINTER) VARIABLE, then under RULES(NOLAXLINK), the compiler would flag an attempt to assign A1 to A2.

- the RETURNS attribute

  For instance if A3 is declared as ENTRY RETURNS(FIXED BIN(31)) and A4 as an ENTRY VARIABLE without the RETURNS attribute, then under RULES(NOLAXLINK), the compiler would flag an attempt to assign A3 to A4.

- the LINKAGE and other OPTIONS suboptions

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

  .

LAXMARGINS | NOLAXMARGINS

Specifying NOLAXMARGINS causes the compiler to flag, depending on the setting of the STRICT and XNUMERIC suboption, lines containing non-blank characters after the right margin. This can be useful in detecting code, such as a closing comment, that has accidentally been pushed out into the right margin.

If the NOLAXMARGINS and STMT options are used together with one of the preprocessors, then any statements that would be flagged because of the NOLAXMARGINS option will be reported as statement zero (since statement numbering occurs only after all the preprocessors are finished, but the detection of text outside the margins occurs as soon as the source is read).

STRICT

Under the STRICT suboption, the compiler will flag any line containing non-blank characters after the right margin

XNUMERIC

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

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 | 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 */
```

LAXSEMI | NOLAXSEMI

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

LAXSTG | NOLAXSTG

Specifying NOLAXSTG causes the compiler to flag declares where a variable A is declared as BASED on ADDR(B) and STG(A) > STG(B) even (and this is the key part) if B is a parameter.

The compiler would already flag this kind of problem if B were in AUTOMATIC or STATIC storage, but it does not, by default, flag this when B is a parameter (since some customers declare B with placeholder attributes that do not describe the actual argument). For those customers whose parameter and argument declares match (or should match), specifying RULES(NOLAXSTG) may help detect more storage overlay problems.

LAXSTRZ | 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 | NOMULTICLOSE

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

UNREF | NOUNREF

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

Default: RULES (IBM BYNAME NODECSIZE EVENDEC ELSEIF GOTO NOLAXBIF NOLAXCTL LAXDCL NOLAXDEF LAXIF LAXINOUT LAXLINK LAXPUNC LAXMARGINS(STRICT) LAXQUAL LAXSEMI LAXSTG NOLAXSTRZ MULTICLOSE UNREF)

SCHEDULER

The SCHEDULER option determines if the compiler tries to optimize instruction scheduling when optimizing a program.

```
       ┌─NOSCHEDULER─┐
►►─────┴─SCHEDULER───┴──────────────────────────────────────►◄
```

Under the SCHEDULER option, the compiler will try to optimize instruction scheduling as part of its optimization efforts.

Under OPT(2) or OPT(3), the SCHEDULER option can increase compile-time by 10% or more.

Under OPT(0), the SCHEDULER has no effect.

SEMANTIC

The SEMANTIC 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.

```
           ┌─NOSEMANTIC─────────────────┐
           │          ┌─────────┐        │
           │          │    ┌─S─┐ │        │
           │          └─(──┼─W─┼─)─┘       │
           │                └─E─┘          │
►►─────────┴─SEMANTIC──────────────────────┴──────────────────────────►◄
```
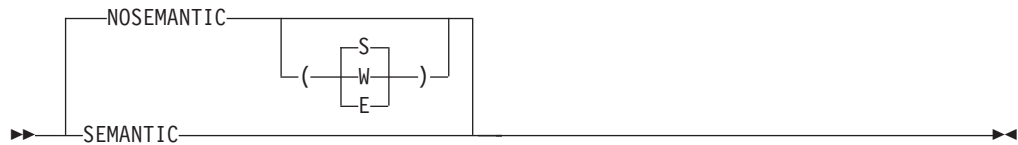
ABBREVIATIONS: SEM, NSEM

SEMANTIC
> Equivalent to NOSEMANTIC(S).

NOSEMANTIC
> Processing stops after syntax checking. No semantic checking is performed.

NOSEMANTIC (S)
> No semantic checking is performed if a severe error or an unrecoverable error has been encountered.

NOSEMANTIC (E)
> No semantic checking is performed if an error, a severe error, or an unrecoverable error has been encountered.

NOSEMANTIC (W)
> No semantic checking is performed if a warning, an error, a severe error, or an unrecoverable error has been encountered.

Semantic checking is not performed if certain kinds of severe errors are found. If the compiler cannot validate that all references resolve correctly (for example, if built-in function or entry references are found with too few arguments) the suitability of any arguments in any built-in function or entry reference is not checked.

SERVICE

The SERVICE option places a string in the object module, if generated. This string is loaded into memory with any load module into which this object is linked, and if the LE dump includes a traceback, this string will be included in that traceback.

```
     ┌─NOSERVICE───────────────────────┐
►►───┴─SERVICE──(──'service string'──)──┴──────────────────────────────►◄
```

ABBREVIATIONS: SERV, NOSERV

The string is limited to 64 characters in length.

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

SOURCE

The SOURCE option specifies that the compiler includes a listing of the source program in the compiler listing. The source program listed is either the original source input or, if any preprocessors were used, the output from the last preprocessor.

```
        ┌─NOSOURCE─┐
►►──────┴─SOURCE───┴──────────────────────────────────────────────────►◄
```

ABBREVIATIONS: S, NS

SPILL

The SPILL option specifies the size of the spill area to be used for the compilation. When too many registers are in use at once, the compiler dumps some of the registers into temporary storage that is called the spill area.

```
►►──SPILL──(size)────────────────────────────────────────────────────►◄
```

ABBREVIATIONS: SP

If you have to expand the spill area, you will receive a compiler message telling you the size to which you should increase it. Once you know the spill area that your source program requires, you can specify the required size (in bytes) as shown in the syntax diagram above. The maximum spill area size is 3900. Typically, you will need to specify this option only when compiling very large programs with OPTIMIZE.

STATIC

The STATIC option controls whether INTERNAL STATIC variables are retained in the object module even if unreferenced.

```
                  ┌─SHORT─┐
►►──STATIC──(─────┴─FULL──┴──)────────────────────────────────────────►◄
```

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.

STDSYS

The STDSYS option specifies that the compiler should cause the SYSPRINT file to be equated to the C stdout file and the SYSIN file to be equated to the C stdin file.

```
        ┌─NOSTDSYS─┐
►►──────┴─STDSYS───┴──────────────────────────────────────────────────►◄
```

Using the STDSYS option may make it easier to develop and debug a mixed PL/I and C application.

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.

```
        ┌─NOSTMT─┐
►►──────┴─STMT───┴──────────────────────────────────────────────────►◄
```

The default is NOSTMT.

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

Note that there is no GOSTMT option. The only option that will produce information at run-time identifying where an error has occurred is the GONUMBER option. Also note that when the GONUMBER option is used, the term ″statement″ in the run-time error messages will refer to line numbers as used by the NUMBER compiler option - even if the STMT option was in effect.

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.

```
        ┌─NOSTORAGE─┐
►►──────┴─STORAGE───┴────────────────────────────────────────────────►◄
```

ABBREVIATIONS: STG, NSTG

The STORAGE output will also include the amount of storage used for the internal static for the compilation.

STRINGOFGRAPHIC

The STRINGOFGRAPHIC option determines whether the result of the STRING built-in function when applied to a GRAPHIC aggregate has the attribute CHARACTER or GRAPHIC.

```
                          ┌─GRAPHIC───┐
►►──STRINGOFGRAPHIC──(────┴─CHARACTER─┴──)───────────────────────────►◄
```

ABBREVIATIONS: CHAR, G

CHARACTER
> Under STRINGOFGRAPHIC(CHAR), if the STRING built-in is applied to an array or structure of UNALIGNED NONVARYING GRAPHIC variables, the result will have the CHARACTER attribute.

GRAPHIC

Under STRINGOFGRAPHIC(GRAPHIC), if the STRING built-in is applied to
an array or structure of GRAPHIC variables, the result will have the GRAPHIC
attribute.

SYNTAX

The SYNTAX option specifies that the compiler continues into syntax checking
after preprocessing when you specify the MACRO option, unless an unrecoverable
error has occurred. Whether the compiler continues with the compilation depends
on the severity of the error, as specified by the NOSYNTAX option.

```
               ┌─NOSYNTAX─────────────────────┐
               │              ┌─S─┐            │
               │         ┌─(───W───)─┐         │
►►─────────────┤         └─────E─────┘         ├─────────────────────────────────►◄
               └─SYNTAX───────────────────────┘
```

ABBREVIATIONS: SYN, NSYN

SYNTAX

Continues syntax checking after preprocessing unless a severe error or an
unrecoverable error has occurred. SYNTAX is equivalent to NOSYNTAX(S).

NOSYNTAX

Processing stops unconditionally after preprocessing.

NOSYNTAX(W)

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

NOSYNTAX(E)

No syntax checking if the compiler detects an error, severe error, or
unrecoverable error.

NOSYNTAX(S)

No syntax checking if the compiler detects a severe error or unrecoverable
error.

If the NOSYNTAX option terminates the compilation, no cross-reference listing,
attribute listing, or other listings that follow the source program is produced.

You can use this option to prevent wasted runs when debugging a PL/I program
that uses the preprocessor.

If the NOSYNTAX option is in effect, any specification of the CICS preprocessor
via the CICS, XOPT or XOPTS options will be ignored. This allows the MACRO
preprocessor to be invoked before invoking the CICS translator.

SYSPARM

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

```
►►─SYSPARM──(──'string' ──)──────────────────────────────────────────────────────►◄
```

string

Can be up to 64 characters long. A null string is the default.

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

SYSTEM

The SYSTEM option specifies the format used to pass parameters to the MAIN PL/I procedure, and generally indicates the host system under which the program runs.

```
►►─SYSTEM─(─┬─MVS──┬─)─────────────────────────────────────────────►◄
            ├─CICS─┤
            ├─IMS──┤
            ├─OS───┤
            └─TSO──┘
```
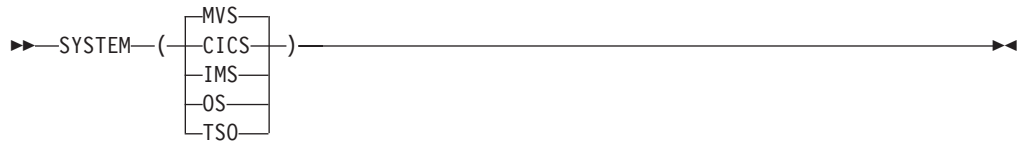
Table 4 shows the type of parameter list you can expect, and how the program runs under the specified host system. It also shows the implied settings of NOEXECOPS. Your MAIN procedure must receive only those types of parameter lists that are indicated as valid in this table. Additional run-time information for the SYSTEM option is provided in *z/OS Language Environment Programming Guide.* .

Table 4. SYSTEM option table

SYSTEM option	Type of parameter list	Program runs as	NOEXECOPS implied
SYSTEM(MVS)	Single varying character string or no parameters.	z/OS application program	NO
	Otherwise, arbitrary parameter list.		YES
SYSTEM(CICS)	Pointer(s)	CICS® transaction	YES
SYSTEM(IMS)	Pointer(s)	IMS™ application program	YES
SYSTEM(OS)	z/OS UNIX parameter list	z/OS UNIX application program	YES
SYSTEM(TSO)	Pointer to CCPL	TSO command processor	YES

Under SYSTEM(IMS), all pointers are presumed to be passed BYVALUE, but under SYSTEM(MVS) they are presumed to be passed BYADDR.

MAIN procedures run under CICS must be compiled with SYSTEM(CICS) or SYSTEM(MVS). If compiled with SYSTEM(MVS), the PTFs for runtime APAR PQ91318 must be applied.

TERMINAL

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

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

```
   ┌─TERMINAL───┐
►►─┴─NOTERMINAL─┴──────────────────────────────────────────────────►◄
```

ABBREVIATIONS: TERM, NTERM

TERMINAL
Messages are displayed on the terminal.

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

TEST

The TEST option specifies the level of testing capability that the compiler generates as part of the object code. It allows you to control the location of test hooks and to control whether or not the symbol table will be generated.

```
                    ┌─NOTEST─────────────────────────┐
                    │          ┌─────,─────┐          │
                    │          │  ┌─ALL──┐ │          │
 ►►─┬──────────────┬┤  TEST─(──┴──┼─STMT─┼─┴──)───┬───┴──►◄
                    │             ├─PATH─┤         │
                    │             ├─BLOCK┤         │
                    │             └─NONE─┘         │
                    │          ┌─SYM───┐           │
                    │          └─NOSYM─┘           │
                    │          ┌─HOOK──┐           │
                    │          └─NOHOOK┘           │
                    │          ┌─NOSEPARATE─┐      │
                    │          └─SEPARATE───┘      │
                    │          ┌─SEPNAME───┐       │
                    │          └─NOSEPNAME─┘       │
                    │          ┌─SOURCE────┐       │
                    │          ├─AFTERALL──┤       │
                    │          ├─AFTERCICS─┤       │
                    │          ├─AFTERMACRO┤       │
                    │          └─AFTERSQL──┘       │
```
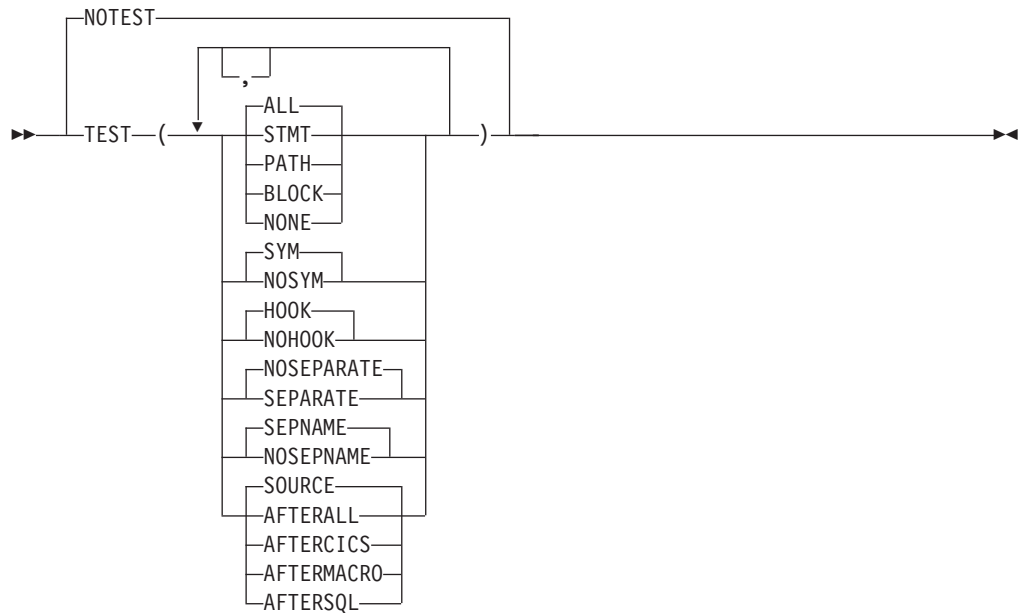
ABBREVIATIONS: AALL, ACICS, AMACRO, ASQL

STMT
Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, inserts hooks at statement boundaries and block boundaries

PATH
Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, tells the compiler to insert hooks:
- Before the first statement enclosed by an iterative DO statement
- Before the first statement of the true part of an IF statement
- Before the first statement of the false part of an IF statement
- Before the first statement of a true WHEN or OTHERWISE statement of a SELECT group
- Before the statement following a user label, excluding labeled FORMAT statements. If a statement has multiple labels, only one hook is inserted.
- At CALLs or function references - both before and after control is passed to the routine
- At block boundaries

BLOCK

Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, tells the compiler to insert hooks at block boundaries (block entry and block exit).

ALL

Causes the compiler to generate a statement table, and if the HOOK suboption is in effect, inserts hooks at all possible locations and generates a statement table.

Note: Under opt(2), hooks are set only at block boundaries.

NONE

No hooks are put into the program.

SYM

Creates a symbol table that allows you to examine variables by name.

NOSYM

No symbol table is generated.

NOTEST

Suppresses the generation of all testing information.

HOOK

Causes the compiler to insert hooks into the generated code if any of the TEST suboptions ALL, STMT, BLOCK or PATH are in effect.

NOHOOK

Causes the compiler not to insert hooks into the generated code.

For Debug Tool to generate overlay hooks, one of the suboptions ALL, PATH, STMT or BLOCK must be specified, but HOOK need not be specified, and NOHOOK would in fact be recommended.

If NOHOOK is specified,ENTRY and EXIT breakpoints are the only PATH breakpoints at which Debug Tool will stop.

SEPARATE

Causes the compiler to place most of the debug information it generates into a separate debug file. Using this option will substantially reduce the size of the object deck created by the compiler when the TEST option is in effect.

If your program contains GET or PUT DATA statements, the separate debug file will contain less debug information since those statements require that symbol table information be placed into the object deck.

The generated debug information will always include a compressed version of the source passed to the compiler. This means that source may be specified via SYSIN DD * or that the source may be a temporary dataset created by an earlier job step (for example, the source may be the output of the old SQL or CICS precompilers).

If this suboption is used in a batch compile, then the JCL for that compile must include a DD card for SYSDEBUG which must name a dataset with RECFM=FB and with 80 <= LRECL <= 1024.

This suboption may not be used with the LINEDIR compiler option.

NOSEPARATE

Causes the compiler to place all of the debug information it generates into the object deck.

Under this option, the generated debug information will not include a compressed version of the source passed to the compiler. This means that source must in a dataset that can be found by DebugTool when you try to debug the program.

SEPNAME

Causes the compiler to place the name of the separate debug file into the object deck.

This option is ignored if the SEPARATE option is not in effect.

NOSEPNAME

Causes the compiler not to place the name of the separate debug file into the object deck.

This option is ignored if the SEPARATE option is not in effect.

SOURCE

Causes the compiler to write the source to the separate debug file, and that source is what will appear in the listing and what Debug Tool will show when it runs.

This option is ignored if the SEPARATE option is not in effect.

AFTERALL

Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the last preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view.

AALL may be used as an abbreviation for AFTERALL

This option is ignored if the SEPARATE option is not in effect.

AFTERCICS

Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the CICS preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view.

ACICS may be used as an abbreviation for AFTERCICS

This option is ignored if the SEPARATE option is not in effect.

AFTERMACRO

Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the MACRO preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view.

AMACRO may be used as an abbreviation for AFTERMACRO

This option is ignored if the SEPARATE option is not in effect.

AFTERSQL

Causes the source listing to show the source as if it came from the MDECK from the last invocation, if any, of the SQL preprocessor, and, more importantly perhaps, it will cause Debug Tool to bring up this as the source view.

ASQL may be used as an abbreviation for AFTERSQL

This option is ignored if the SEPARATE option is not in effect.

Note:

- Under opt(2) or opt(3), hooks are set only at block boundaries.

- You must use Debug Tool Version 6 (or later) to debug code compiled with the SEPARATE compiler option.
- There is no support for an input file that spans concatenated datasets.

Specifying TEST(NONE,NOSYM) causes the compiler to set the option to NOTEST.

Use of TEST(NONE,SYM) is strongly discouraged, and it is unclear what is intended when you specify these settings. You would probably be much better off if you specified TEST(ALL,SYM,NOHOOK) or TEST(STMT,SYM,NOHOOK).

Any TEST option other than NOTEST and TEST(NONE,NOSYM) will automatically provide the attention interrupt capability for program testing.

If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following:
- The INTERRUPT option
- A TEST option other than NOTEST or TEST(NONE,NOSYM)

Note: ATTENTION is supported only under TSO.

The TEST option will imply GONUMBER.

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

If the TEST option is specified, no inlining will occur.

Structures with REFER are supported in the symbol table.

If TEST(SYM) is in effect, the compiler will generate tables to enable the automonitor feature of DebugTool. These tables may substantially increase the size of the object module unless the TEST(SEPARATE) option is in effect. When the automonitor feature of DebugTool is activated, these tables will be used to display the values of the variables used in a statement before the statement executes - as long as the variable has computational type or has the attribute POINTER, OFFSET or HANDLE. If the statement is an assignment statement, the value of the target will also be displayed; however, if the target has not been initialized or assigned previously, its value will be meaningless.

Any variable declared with an * for its name will not be visible when using DebugTool. Additionally, if an * is used as the name of a parent structure or substructure, then all of its children will also be invisible. For these purposes, it might be better to use a single underscore for the name of any structure elements that you wish to leave "unnamed".

As an example of the differing effects of the AFTERMACRO, AFTERSQL and AFTERALL suboptions suppose the PP option was PP(MACRO('INCONLY'), SQL, MACRO). Then:
- Under TEST(SEP,AFTERMACRO), the "source" in the listing and in the Debug Tool source window would appear as if it came from the MDECK that the second invocation of the MACRO preprocessor would have produced
- Under TEST(SEP,AFTERSQL), the "source" in the listing and in the Debug Tool source window would appear as if it came from the MDECK that the invocation of the SQL preprocessor would have produced (and hence %DCL and other macro statements would still be visible)

- Under TEST(SEP,AFTERALL), the ″source″ would be as under the TEST(SEP,AFTERMACRO) option since the MACRO preprocessor is the last in the PP option

TUNE

The TUNE option specifies the architecture for which the executable program will be optimized. This option allows the optimizer to take advantage of architectural differences such as scheduling of instructions.

```
►►──TUNE──(──┬──5──┬──)──────────────────────────────────►◄
             └──n──┘
```

Note: If TUNE level is lower than ARCH, TUNE is forced to ARCH.

The current values that may be specified for the TUNE level are:

0 Generates code that is executable on all models, but that does not take advantage of any architectural differences.

1 Generates code that is executable on all models but is optimized for the models specified under ARCH(1).

2 Generates code that is executable on all models but is optimized for the models specified under ARCH(2).

3 Generates code that is executable on all models but is optimized for the models specified under ARCH(3).

4 Generates code that is executable on all models but is optimized for the models specified under ARCH(4).

5 Generates code that is executable on all models but is optimized for the models specified under ARCH(5).

6 Generates code that is executable on all models but is optimized for the models specified under ARCH(6).

7 Generates code that is executable on all models but is optimized for the models specified under ARCH(7).

If you specify a TUNE value less than 5, the compiler will reset it to 5.

USAGE

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

```
                    ┌──,──┐
►►──USAGE──(──▼──┬──ROUND──(──┬──IBM──┬──)──┬──)──────────────►◄
                 │            └──ANS──┘      │
                 └──UNSPEC──(──┬──IBM──┬──)──┘
                              └──ANS──┘
```

ROUND(IBM | ANS)
Under the ROUND(IBM) suboption, the second argument to the ROUND built-in function is ignored if the first argument has the FLOAT attribute.

Under the ROUND(ANS) suboption, the ROUND built-in function is implemented as described in the *PL/I Language Reference.*

UNSPEC(IBM | ANS)
Under the UNSPEC(IBM) suboption, UNSPEC cannot be applied to a structure and, if applied to an array, returns an array of bit strings.

Under the UNSPEC(ANS) suboption, UNSPEC can be applied to structures and, when applied to a structure or an array, UNSPEC returns a single bit string.

Default: USAGE(ROUND(IBM) UNSPEC(IBM))

WIDECHAR

The WIDECHAR option specifies the format in which WIDECHAR data will be stored.

```
►►─WIDECHAR─(─┬─BIGENDIAN────┬─)──────────────────────────►◄
             └─LITTLEENDIAN─┘
```

BIGENDIAN
Indicates that WIDECHAR data will be stored in big-endian format. For instance, the WIDECHAR value for the UTF-16 character '1' will be stored as '0031'x.

LITTLEENDIAN
Indicates that WIDECHAR data will be stored in little-endian format. For instance, the WIDECHAR value for the UTF-16 character '1' will be stored as '3100'x.

WX constants should always be specified in big-endian format. Thus the value '1' should always be specified as '0031'wx, even if under the WIDECHAR(LITTLEENDIAN) option, it is stored as '3100'x.

WINDOW

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

```
►►─WINDOW─(─┬─1950─┬─)────────────────────────────────────►◄
           └─w────┘
```

w   Either an unsigned integer that represents the start of a fixed window or a negative integer that specifies a "sliding" window. For example, `WINDOW(-20)` indicates a window that starts 20 years prior to the year when the program runs.

WRITABLE

The WRITABLE option specifies that the compiler may treat static storage as writable (and if it does, this would make the resultant code non-reentrant).

```
►►─┬─WRITABLE────────────────────┬────────────────────────►◄
   └─NOWRITABLE─┬──────────────┬─┘
               └─(─┬─FWS─┬─)─┘
                   └─PRV─┘
```
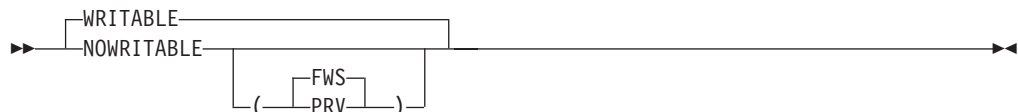
This option has no effect on programs compiled with the RENT option.

The NORENT WRITABLE options allow the compiler to use a static pointer
- as the base for the stack that tracks a CONTROLLED variable
- as the handle for the storage that represents a FILE

So, under the NORENT WRITABLE options, a module using CONTROLLED variables or performing I/O would not be reentrant.

The NORENT NOWRITABLE options require the compiler not to use a static pointer
- as the base for the stack that tracks a CONTROLLED variable
- as the handle for the storage that represents a FILE

So, under the NORENT NOWRITABLE options, a module using CONTROLLED variables or performing I/O will be reentrant.

The FWS and PRV suboptions determine how the compiler handles CONTROLLED variables:

FWS

Upon entry to an EXTERNAL procedure, the compiler makes a library call to find storage it can use to address the CONTROLLED variables in that procedure (and any subprocedures).

PRV

The compiler will use the same pseudoregister variable mechanism used by the old OS PL/I compiler to address CONTROLLED variables.

Hence, under the NORENT NOWRITABLE(PRV) options, old and new code may share CONTROLLED variables.

However, this also means that under the NORENT NOWRITABLE(PRV) options, the use of CONTROLLED variables is subject to all the restrictions as under the old compiler.

Under the NORENT NOWRITABLE(FWS) options, an application may not perform as well as if it were compiled with the RENT or WRITABLE options if the application
- uses CONTROLLED variables
- assigns FILE CONSTANTs to FILE VARIABLEs

The performance of an application under NORENT NOWRITABLE(FWS) may be especially bad if it uses many CONTROLLED variables in many PROCEDUREs.

Under the NOWRITABLE option, the following may not be declared in a PACKAGE outside a PROCEDURE:
- CONTROLLED variables
- FETCHABLE ENTRY constants
- FILE constants

Code compiled with NORENT WRITABLE cannot be mixed with code compiled with NORENT NOWRITABLE if they share any external CONTROLLED variables. In general, you should avoid mixing code compiled with WRITABLE with code compiled with NOWRITABLE.

XINFO

The XINFO option specifies that the compiler should generate additional files with extra information about the current compilation unit.

```
                        ┌─,─┐
                        │   │
            ┌───────────┴───┴───────────┐
►►──XINFO──(──▼──┬─NODEF─┬──────────────)─────────────────────────────►◄
                 ├─DEF───┤
                 ├─NOMSG─┤
                 ├─MSG───┤
                 ├─NOSYM─┤
                 ├─SYM───┤
                 ├─NOSYN─┤
                 ├─SYN───┤
                 ├─NOXMI─┤
                 ├─XMI───┤
                 ├─NOXML─┤
                 └─XML───┘
```
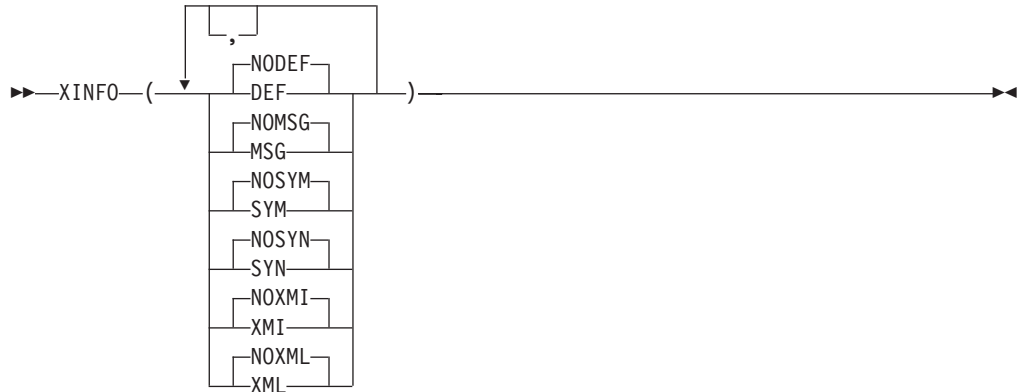
DEF

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 z/OS 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.

MSG

Message information is generated to the ADATA file. See the appendix for more details on the format of the ADATA file.

Under batch, the ADATA file is generated to the file specified by the SYSADATA DD statement. Under z/OS UNIX, the ADATA is generated in the same directory as the object file and has an extension of adt.

NOMSG

No message information is generated to the ADATA file. If neither MSG nor SYM is specified, no ADATA file is generated.

SYM

Symbol information is generated to the ADATA file. See the appendix for more details on the format of the ADATA file.

Under batch, the ADATA file is generated to the file specified by the SYSADATA DD statement. Under z/OS UNIX, the ADATA file is generated in the same directory as the object file and has en extension of adt.

NOSYM

No symbol information is generated to the ADATA file.

SYN

Syntax information is generated to the ADATA file. See the appendix for more details on the format of the ADATA file. Specifying the XINFO(SYN) option can greatly increase the amount of storage, both in memory and for the file produced, required by the compiler.

Under batch, the ADATA file is generated to the file specified by the SYSADATA DD statement. Under z/OS UNIX, the ADATA file is generated in the same directory as the object file and has en extension of adt.

NOSYN

No syntax information is generated to the ADATA file.

XMI

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

Under batch, this file is written to the file specified by the SYSXMI DD statement. Under z/OS UNIX Systems Services, this file is written to the same directory as the object deck and has the extension ″xmi″.

NOXMI

No XMI side-file is created.

XML

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

Under batch, this file is written to the file specified by the SYSXMLSD DD statement. Under z/OS 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.

XML

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

```
>>──XML──(──CASE──(──┬──UPPER──┬──)──)──────────────────────────────><
                     └──ASIS───┘
```

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

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

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.

```
        ┌──NOXREF──────────────────────────┐
>>──┬───┴──XREF──┬──────────────────────┬──┴──────────────────────────><
                 │     ┌──FULL──┐        │
                 └──(──┴──SHORT──┴──)─────┘
```

ABBREVIATIONS: X, NX

FULL
Includes all identifiers and attributes in the compiler listing.

SHORT
Omits unreferenced identifiers from the compiler listing.

The only names not included in the cross reference listing created when using the XREF option are label references on END statements. For example, assume that

statement number 20 in the procedure PROC1 is END PROC1;. In this situation, statement number 20 does not appear in the cross reference listing for PROC1.)

If you specify both the XREF and ATTRIBUTES options, the two listings are combined. If there is a conflict between SHORT and FULL, the usage is determined by the last option specified. For example, ATTRIBUTES(SHORT) XREF(FULL) results in the FULL option for the combined listing.

For a description of the format and content of the cross-reference table, see "Cross-reference table" on page 81.

Blanks, comments and strings in options

Wherever you may use a blank when specifying an option, you may also specify as many blanks or comments as you wish.

However, if a comment is specified in %PROCESS line or in a line in an options file, then the comment must end on the same line as which it begins.

Similarly, if a comment starts in the command line or the PARM= specification, then it must also end there.

The same rule applies to strings: if a string is specified in %PROCESS line or in a line in an options file, then the string must end on the same line as which it begins. Similarly, if a string starts in the command line or the PARM= specification, then it must also end there.

Changing the default options

If you want to change the supplied default compiler options, during installation of the compiler you should edit and submit sample job IBMZWIOP.

This job will let you specify options that will be applied before any other options, thus effectively changing the default options. This job will also let you specify options that will be applied after all other options, thus effectively changing the default options and preventing them from being overridden.

If you want to change the defaults for the macro preprocessor options, you can also do this at installation time by specifying the appropriate PPMACRO option as part of this job. The PPCICS and PPSQL options let you make the corresponding changes for the CICS and SQL preprocessors respectively.

Consult the instructions in the sample job for more information.

Specifying options in the %PROCESS or *PROCESS statements

You can use either %PROCESS or *PROCESS in your program; they are equally acceptable. For consistency and readability in this book, we will always refer to %PROCESS.

The %PROCESS statement identifies the start of each external procedure and allows compiler options to be specified for each compilation. The options you specify in adjacent %PROCESS statements apply to the compilation of the source statements to the end of input, or the next %PROCESS statement.

To specify options in the %PROCESS statement, code as follows:

```
%PROCESS options;
```

where *options* is a list of compiler options. You must end the list of options with a semicolon, and the options list should not extend beyond the default right-hand source margin. The percent sign (%) or asterisk (*) must appear in the first column of the record. The keyword PROCESS can follow in the next byte (column) or after any number of blanks. You must separate option keywords by a comma or at least one blank.

The number of characters is limited only by the length of the record. If you do not wish to specify any options, code:

```
%PROCESS;
```

If you find it necessary to continue the %PROCESS statement onto the next record, terminate the first part of the list after any delimiter, and continue on the next record. You cannot split keywords or keyword arguments across records. You can continue a %PROCESS statement on several lines, or start a new %PROCESS statement. An example of multiple adjacent %PROCESS statements is as follows:

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST TEST ;
```

Compile-time options, their abbreviated syntax, and their IBM-supplied defaults are shown in Table 3 on page 3.

How the compiler determines if there are any %PROCESS statements depends on the format of the initial source file:

F or FB format
> if the first character in the record is a ″*″ or a ″%″, then the compiler will check if the next non-blank characters are ″PROCESS″.

V or VB format
> if the first character in the record is a numeric, then the compiler will assume that the first 8 characters are sequence numbers, and if the ninth character is a ″*″ or a ″%″, then it will check if the next non-blank characters are ″PROCESS″. However, if the first character is not a numeric, then if the first character is a ″*″ or ″%″, the compiler will check if the next non-blank characters are ″PROCESS″.

U format
> if the first character in the record is a ″*″ or a ″%″, then the compiler will check if the next non-blank characters are ″PROCESS″.

Using % statements

Statements that direct the operation of the compiler begin with a percent (%) symbol. % statements allow you to control the source program listing and to include external strings in the source program. % statements must not have label or condition prefixes and cannot be a unit of a compound statement. You should place each % statement on a line by itself.

The usage of each % control statement—%INCLUDE, %PRINT, %NOPRINT, %OPTION, %PAGE, %POP, %PUSH, and %SKIP—is listed below. For a complete description of these statements, see *PL/I Language Reference.*

%INCLUDE
> Directs the compiler to incorporate external strings of characters and/or graphics into the source program.

%PRINT

Directs the compiler to resume printing the source and insource listings.

%NOPRINT

Directs the compiler to suspend printing the source and insource listings until a %PRINT statement is encountered.

%OPTION

Specifies one of a selected subset of compiler options for a segment of source code.

%PAGE

Directs the compiler to print the statement immediately after a %PAGE statement in the program listing on the first line of the next page.

%POP Directs the compiler to restore the status of the %PRINT, %NOPRINT, and %OPTION saved by the most recent %PUSH.

%PUSH

Saves the current status of the %PRINT, %NOPRINT, and %OPTION in a *push down* stack on a last-in, first-out basis.

%SKIP Specifies the number of lines to be skipped.

Using the %INCLUDE statement

%INCLUDE statements are used to include additional PL/I files at specified points in a compilation unit. The *PL/I Language Reference* describes how to use the %INCLUDE statement to incorporate source text from a library into a PL/I program.

For a batch compile

A *library* is an z/OS partitioned data set that can be used to store other data sets called members. Source text that you might want to insert into a PL/I program using a %INCLUDE statement must exist as a member within a library. "Source Statement Library (SYSLIB)" on page 138 further describes the process of defining a source statement library to the compiler.

The statement:

```
%INCLUDE DD1 (INVERT);
```

specifies that the source statements in member INVERT of the library defined by the DD statement with the name DD1 are to be inserted consecutively into the source program. The compilation job step must include appropriate DD statements.

If you omit the ddname, the ddname SYSLIB is assumed. In such a case, you must include a DD statement with the name SYSLIB. (The IBM-supplied cataloged procedures do not include a DD statement with this name in the compilation procedure step.)

For a z/OS UNIX compile

The name of the actual include file must be lowercase, unless you specify UPPERINC. 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 looks for INCLUDE files in the following order:
1. Current directory

2. Directories specified with the –I flag or with the INCDIR compiler option
3. /usr/include directory
4. PDS specified with the INCPDS compiler option

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

A %PROCESS statement in source text included by a %INCLUDE statement results in an error in the compilation.

Figure 1 shows the use of a %INCLUDE statement to include the source statements for FUN in the procedure TEST. The library HPU8.NEWLIB is defined in the DD statement with the qualified name PLI.SYSLIB, which is added to the statements of the cataloged procedure for this job. Since the source statement library is defined by a DD statement with the name SYSLIB, the %INCLUDE statement need not include a ddname.

It is not necessary to invoke the preprocessor if your source program, and any text to be included, does not contain any macro statements.

```
//OPT4#9     JOB
//STEP3      EXEC IBMZCBG,PARM.PLI='INC,S,A,X,NEST'
//PLI.SYSLIB DD DSN=HPU8.NEWLIB,DISP=OLD
//PLI.SYSIN  DD *
   TEST: PROC OPTIONS(MAIN) REORDER;
     DCL ZIP PIC '99999';            /* ZIP CODE                    */
     DCL EOF BIT INIT('0'B);
     ON ENDFILE(SYSIN) EOF = '1'B;
     GET EDIT(ZIP) (COL(1), P'99999');
     DO WHILE(¬EOF);
       PUT SKIP EDIT(ZIP, CITYFUN(ZIP)) (P'99999', A(16));
       GET EDIT(ZIP) (COL(1), P'99999');
     END;
     %PAGE;
     %INCLUDE FUN;
   END;                             /* TEST                        */
//GO.SYSIN DD *
95141
95030
94101
//
```

Figure 1. Including source statements from a library

Using the %OPTION statement

With the %OPTION statement, you can change the setting of selected compiler options. The change remains in effect until the next %POP; statement is read.

Currently, you may use the %OPTION statement only to switch the LANGLVL option from SAA2 to SAA (or vice versa). So its syntax is:

```
►►──%OPTION──LANGLVL──(──┬──SAA───┬──)──────────────────────────────►◄
                         └─SAA2──┘
```

For example, if you had some code that was using a LANGLVL(SAA2) feature in a compilation for which you have specified the LANGLVL(SAA) option, you could enclose make the compiler accept that code by taking the following steps:

1. insert a %PUSH; statement
2. insert a %OPTION LANGLVL(SAA2) statement
3. insert the code using LANGLVL(SAA2) features
4. insert a %POP; statement;

Using the compiler listing

During compilation, the compiler generates a listing, most of which is optional, that contains information about the source program, the compilation, and the object module. The following description of the listing refers to its appearance on a printed page.

Note:

> Although the compiler listing is for your use, it is not a programming interface and is subject to change.

Of course, if compilation terminates before reaching a particular stage of processing, the corresponding listings do not appear.

Heading information

The first page of the listing is identified by the product number, the compiler version number, a string specifying when the compiler was built, and the date and the time compilation commenced. This page and subsequent pages are numbered.

The listing will then show any options that have been specified for this compilation. These options will be shown even if the NOOPTIONS option has been specified and will include, in order, the following:

- the initial install options (those are the options set at install time and which are applied before any other options)
- under z/OS UNIX, any options specified in the IBM_OPTIONS environment variable
- any options specified in the parameter string passed to the compiler (i.e. in the command line under z/OS UNIX or in the PARM= under batch)
- any options specified in options files named in the compiler parameter string

  This will include the name of each options file and its contents in the same form as read by the compiler.
- any options specified in *PROCESS or %PROCESS lines in the source
- the final install options (those are the options set at install time and which are applied after any other options)

Near the end of the listing you will find either a statement that no errors or warning conditions were detected during the compilation, or a message that one or more errors were detected. The format of the messages is described under "Messages and return codes" on page 85. The second to the last line of the listing shows the time taken for the compilation. The last line of the listing is *END OF COMPILATION OF xxxx.* where *xxxx* is the external procedure name. If you specify the NOSYNTAX compiler option, or the compiler aborts early in the compilation, the external procedure name *xxxx* is not included and the line truncates to *END OF COMPILATION.*

The following sections describe the optional parts of the listing in the order in which they appear.

Options used for compilation

If you specify the OPTIONS option, a complete list of the options specified for the compilation, including the default options, appears on the following pages. It will show the settings of all the options finally in effect during the compilation. If the setting of an option differs from the default setting after the initial install options were applied, then that line will be marked with a +

Preprocessor input

If you specify both the MACRO and INSOURCE options, the compiler lists input to the preprocessor, one record per line, each line numbered sequentially at the left.

If the preprocessor detects an error, or the possibility of an error, it prints a message on the page or pages following the input listing. The format of these messages is the same as the format for the compiler messages described under "Messages and return codes" on page 85.

SOURCE program

If you specify the SOURCE option, the compiler lists one record per line. These records will always include the source line and file numbers. However, if a file contains 999,999 or more lines, then the compiler will flag the file as too large and will list only the last 6 digits in the source line numbers for that file.

If the input records contain printer control characters, or %SKIP or %PAGE statements, the lines are spaced accordingly. Use %NOPRINT and %PRINT statements to stop and restart the printing of the listing.

If you specify the MACRO option, the source listing shows the included text in place of the %INCLUDE statements in the primary input data set.

Statement nesting level

If you specify the NEST option, the block level and the DO-level are printed to the right of the statement or line number under the headings LEV and NT respectively, as in the following example:

```
Line.File LV NT
   1.0            A: PROC OPTIONS(MAIN);
   2.0     1      B: PROC;
   3.0     2         DCL K(10,10) FIXED BIN (15);
   4.0     2         DCL Y FIXED BIN (15) INIT (6);
   5.0     2         DO I=1 TO 10;
   6.0     2 1          DO J=1 TO 10;
   7.0     2 2             K(I,J) = N;
   8.0     2 2          END;
   9.0     2 1       BEGIN;
  10.0     3 1          K(1,1)=Y;
  11.0     3 1       END;
  12.0     2 1    END B;
  13.0     1      END A;
```

ATTRIBUTE and cross-reference table

If you specify the ATTRIBUTES option, the compiler prints an attribute table containing a list of the identifiers in the source program together with their declared and default attributes.

If you specify the XREF option, the compiler prints a cross-reference table containing a list of the identifiers in the source program together with the file and line numbers of the statements in which they appear.

If you specify both ATTRIBUTES and XREF, the two tables are combined. In these tables, if you explicitly declare an identifier, the compiler will list file number and line number of its DECLARE. Contextually declared variables are marked by +++++, and other implicitly declared variables are marked by *****.

Attribute table

The compiler never includes the attributes INTERNAL and REAL. You can assume them unless the respective conflicting attributes, EXTERNAL and COMPLEX, appear.

For a file identifier, the attribute FILE always appears, and the attribute EXTERNAL appears if it applies; otherwise, the compiler lists only explicitly declared attributes.

The OPTIONS attribute will not appear unless the ENTRY attribute applies, and then only the following options would appear (as appropriate)
- ASSEMBLER
- COBOL
- FETCHABLE
- FORTRAN
- NODESCRIPTOR
- RETCODE

The compiler prints the dimension attribute for an array first. It prints the bounds as in the array declaration, but expressions are replaced by asterisks unless they have been reduced by the compiler to a constant, in which case the value of the constant is shown.

For a character string, a bit string, a graphic string, or an area variable, the compiler prints the length, as in the declaration, but expressions are replaced by asterisks unless they have been reduced by the compiler to a constant, in which case the value of the constant is shown.

Cross-reference table

If you combine the cross-reference table with the attribute table, the list of attributes for a name is identified by file number and line number. 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 you specify ATTRIBUTES and XREF, the two tables are combined.

If there are unreferenced identifiers, they are displayed in a separate table.

Aggregate length table

An aggregate length table is obtained by using the AGGREGATE option. The table includes structures but not arrays that have non-constant extents, but the sizes and

offsets of elements within structures with non-constant extents may be inaccurate or specified as *. For the aggregates listed, the table contains the following information:

- Where the aggregate is declared.
- The name of the aggregate and each element within the aggregate.
- The byte offset of each element from the beginning of the aggregate.
- The length of each element.
- The total length of each aggregate, structure, and substructure.
- The total number of dimensions for each element.

Please be careful when interpreting the data offsets indicated in the data length table. An odd offset does not necessarily represent a data element without halfword, fullword, or even double word alignment. If you specify or infer the aligned attribute for a structure or its elements, the proper alignment requirements are consistent with respect to other elements in the structure, even though the table does not indicate the proper alignment relative to the beginning of the table.

If there is padding between two structure elements, a /*PADDING*/ comment appears, with appropriate diagnostic information.

Statement offset addresses

If the LIST compile option is used, the compiler includes a pseudo-assembler listing in the compiler listing. This listing includes, for each instruction, an offset whose meaning depends on the setting of the BLKOFF compiler option:

- Under the BLKOFF option, this offset is the offset of the instruction from the primary entry point for the function or subroutine to which it belongs. Thus under this option, the offsets are reset with each new block.
- Under the NOBLKOFF option, this offset is the offset of the instruction from the start of the compilation unit. Thus under this option, the offsets are cumulative.

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

These offsets can be used with the offset given in a run-time error message to determine the statement to which the message applies.

The OFFSET option produces a table that gives for each statement, the offset of the first instruction belonging to that statement.

In the example shown in Figure 2 on page 83, the message indicates that the condition was raised at offset +58 from the SUB1 entry. The compiler listing excerpt shows this offset associated with line number 8. The run-time output from this erroneous statement is shown if Figure 3 on page 83.

```
Compiler Source
  Line.File
      2.0       TheMain: proc options( main );
      3.0         call sub1();
      4.0         Sub1: proc;
      5.0           dcl (i, j) fixed bin(31);
      6.0
      7.0           i = 0;
      8.0           j = j / i;
      9.0         end Sub1;
     10.0       end TheMain;

 . . .

OFFSET OBJECT CODE         LINE# FILE#   P S E U D O   A S S E M B L Y   L I S T I N G
000000                     00002 |       THEMAIN DS    0D

 . . .

00004C  5800  C1F4         00002 |               L     r0,_CEECAA_(,r12,500)
000050  5000  D098         00002 |               ST    r0,#_CEECAACRENT_1(,r13,152)
000054  5810  D098         00000 |               L     r1,#_CEECAACRENT_1(,r13,152)
000058  5820  3062         00000 |               L     r2,=Q(@STATIC)(,r3,98)
00005C  4152  1000         00000 |               LA    r5,=Q(@STATIC)(r2,r1,0)
000060  18BD               00003 |               LR    r11,r13
000062  5800  D098         00003 |               L     r0,#_CEECAACRENT_1(,r13,152)
000066  5000  C1F4         00003 |               ST    r0,_CEECAA_(,r12,500)
00006A  58F0  3066         00003 |               L     r15,=A(SUB1)(,r3,102)
00006E  05EF               00003 |               BALR  r14,r15
000070                     00010 |       @1L1    DS    0H
000070  5810  5000         00010 |               L     r1,IBMQEFSH(,r5,0)
000074  58F0  1008         00010 |               L     r15,&Func_&WSA(,r1,8)
000078  5800  100C         00010 |               L     r0,&Func_&WSA(,r1,12)
00007C  5000  C1F4         00010 |               ST    r0,_CEECAA_(,r12,500)
000080  05EF               00010 |               BALR  r14,r15
000082                     00010 |       @1L4    DS    0H
000082  5800  D098         00002 |               L     r0,#_CEECAACRENT_1(,r13,152)
000086  5000  C1F4         00002 |               ST    r0,_CEECAA_(,r12,500)

 . . .

000000                     00004 |       SUB1    DS    0D

 . . .

000048  4100  0000         00007 |               LA    r0,0
00004C  5000  D098         00007 |               ST    r0,I(,r13,152)
000050  5840  D09C         00008 |               L     r4,J(,r13,156)
000054  8E40  0020         00008 |               SRDA  r4,32
000058  1D40               00008 |               DR    r4,r0
00005A  1805               00008 |               LR    r0,r5
00005C  5000  D09C         00008 |               ST    r0,J(,r13,156)
```

Figure 2. Finding statement number (compiler listing example)

```
Message :

IBM0301S ONCODE=320   The ZERODIVIDE condition was raised.
        From entry point SUB1 at compile unit offset +00000058 at
        address 0D3012C0.
```

Figure 3. Finding statement number (run-time message example)

Entry offsets given in dump and ON-unit SNAP error messages can be compared with this table and the erroneous statement discovered. The statement is identified by finding the section of the table that relates to the block named in the message and then finding the largest offset less than or equal to the offset in the message. The statement number associated with this offset is the one needed.

Storage offset listing

If the MAP compile option is used, the compiler includes a storage offset listing in the compiler listing. This listing gives the location in storage of the following level-1 variables if they are used in the program:

- AUTOMATIC
- CONTROLLED except for PARAMETERs
- STATIC except for ENTRY CONSTANTs that are not FETCHABLE

The listing may also include some compiler generated temporaries.

For an AUTOMATIC variable with adjustable extents, there will be two entries in this table:

- an entry with '_addr' prefixing the variable name - this entry gives the location of the address of the variable
- an entry with '_desc' prefixing the variable name - this entry gives the location of the address of the variable's descriptor

For STATIC and CONTROLLED variables, the storage location will depend on the RENT/NORENT compiler option, and if the NORENT option is in effect, the location of CONTROLLED variables will also depend on the WRITABLE/NOWRITABLE compiler option.

The first column in the Storage Offset Listing is labeled *IDENTIFIER* and holds the name of the variable whose location appears in the fourth column.

The second column in the Storage Offset Listing is labeled *DEFINITION* and holds a string in the format ″B-F:N″ where

- *B* is the number of the block where the variable is declared

  You can find the name of the block corresponding to this block number in the Block Name List which will proceed the Storage Offset Listing (and the Pseudo Assembly Listing, if any)
- *F* is the number of the source file where the variable is declared

  You can find the name of the file corresponding to this file number in the File Reference Table which will appear very near the end of the entire compilation listing.
- *N* is the number of the source line where the variable is declared in that source file

The third column in the Storage Offset Listing is labeled *ATTRIBUTES* and indicates the storage class of the variable.

The fourth column in the Storage Offset Listing is unlabeled and tells how to find the location of the variable.

This storage offset listing is sorted by block and by variable name, and it also includes only user variables. However, specifying the MAP option also causes the compiler to produce

- a "static map" which lists all STATIC variables but sorted by hex offset
- an "automatic map" which lists, for each block, all AUTOMATIC variables but sorted by hex offset

File reference table

The file reference table consists of three columns which list the following information about the files read during the compile:

- the number assigned by the compiler to the file
- the included-from data for the file
- the name of the file

The first entry in the included-from column is blank because the first file listed is the source file. Subsequent entries in this column show the line number of the include statement followed by a period and the file number of the source file containing the include.

If the file is a member of a PDS or PDSE, the file name lists the fully qualified dataset name and the member name.

If the file is included via a subsystem (such as Librarian), then the file name will have the form *DD:ddname(member)*, where

- *ddname* is the ddname specified on the %INCLUDE statement (or SYSLIB if no ddname was specified)
- *member* is the member name specified on the %INCLUDE statement.

Messages and return codes

If the preprocessor or the compiler detects an error, or the possibility of an error, messages are generated. Messages generated by the preprocessor appear in the listing immediately after the listing of the statements processed by the preprocessor. You can generate your own messages in the preprocessing stage by use of the %NOTE statement. Such messages might be used to show how many times a particular replacement had been made. Messages generated by the compiler appear at the end of the listing.

For compilations that produce no messages, the compiler will include a line saying "no compiler messages" where the compiler messages would have been listed.

Messages are displayed in the following format:

```
PPPnnnnI X
```

where `PPP` is the prefix identifying the origin of the message (for example, IBM indicates the PL/I compiler), `nnnn` is the 4-digit message number, and `X` identifies the severity code. All messages are graded according to their severity, and the severity codes are I, W, E, S, and U.

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved. For z/OS, this code appears in the *end-of-step* message that follows the listing of the job control statements and job scheduler messages for each step.

Table 5 on page 86 provides an explanation of the severity codes and the comparable return code for each:

Table 5. Description of PL/I error codes and return codes

Severity Code	Return Code	Message Type	Description
I	0000	Informational	The compiled program should run correctly. The compiler might inform you of a possible inefficiency in your code or some other condition of interest.
W	0004	Warning	A statement might be in error (warning) even though it is syntactically valid. The compiled program should run correctly, but it might produce different results than expected or be significantly inefficient.
E	0008	Error	A simple error fixed by the compiler. The compiled program should run correctly, but it might product different results than expected.
S	0012	Severe	An error not fixed by the compiler. If the program is compiled and an object module is produced, it should not be used.
U	0016	Unrecoverable	An error that forces termination of the compilation. An object module is not successfully created.

Note: Compiler messages are printed in groups according to these severity levels.

The compiler lists only messages that have a severity equal to or greater than that specified by the FLAG option, as shown in Table 6.

Table 6. Using the FLAG option to select the lowest message severity listed

Type of Message	Option
Information	FLAG(I)
Warning	FLAG(W)
Error	FLAG(E)
Severe Error	FLAG(S)
Unrecoverable Error	Always listed

The text of each message, an explanation, and any recommended programmer response, are given in *Enterprise PL/I Messages and Codes*.

Chapter 2. PL/I preprocessors

The PL/I compiler allows you to select one or more of the integrated preprocessors as required for use in your program. You can select the include preprocessor, the macro preprocessor, 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 preprocessor, based on %statements and macros, modifies your source program.
- The SQL preprocessor modifies your source program and translates EXEC SQL statements into PL/I statements.
- The CICS preprocessor modifies your source program and translates EXEC CICS statements into PL/I statements.

Each preprocessor supports a number of options to allow you to tailor the processing to your needs.

The three compile-time options MDECK, INSOURCE, and SYNTAX are meaningful only when you also specify the PP option. For more information about these options, see "MDECK" on page 42, "INSOURCE" on page 34, and "SYNTAX" on page 63.

87

Include preprocessor

The include preprocessor allows you to incorporate external source files into your programs by using include directives other than the PL/I directive %INCLUDE.

The following syntax diagram illustrates the options supported by the INCLUDE preprocessor:

```
►►─PP─(─INCLUDE─(─'─ID(<directive>)─'─)─)─────────────────────────►◄
```

ID Specifies the name of the include directive. Any line that starts with this directive as the first set of nonblank characters is treated as an include directive.

The specified directive must be followed by one or more blanks, an include member name, and finally an optional semicolon. Syntax for ddname(membername) is not supported.

In the following example, the first include directive is valid and the second one is not:

```
++include payroll
++include syslib(payroll)
```

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)'))
```

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*.

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.

```
►►──FIXED──(──┬─DECIMAL─┬──)──────────────────────────────────►◄
              └─BINARY──┘
```

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.

```
►►──CASE──(──┬─UPPER─┬──)──────────────────────────────────────►◄
            └─ASIS──┘
```

ASIS

the input text is left ″as is″.

UPPER

the input text is to be converted to upper case.

INCONLY

This option specifies that the preprocessor should process only %INCLUDE

and %XINCLUDE statements. When this option is in effect, you may use neither INCLUDE or XINCLUDE as a macro

- procedure name
- statement label
- variable name

NOINCONLY

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

RESCAN

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

```
►►──RESCAN──(──┬─ASIS──┬──)──────────────────────────►◄
              └─UPPER─┘
```

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.

DBCS

This option specifies if the preprocessor should normalize DBCS during text replacement.

```
        ┌─INEXACT─┐
►►──DBCS──(──┴─EXACT───┴──)──────────────────────────────────────────►◄
```

EXACT

the input text is left ″as is″, and the preprocessor will treat <kk.B> and <kk>B as different names.

INEXACT

the input text is ″normalized″, and the preprocessor will treat <kk.B> and <kk>B as two versions of the same name.

Macro preprocessor example

A simple example of the use of the preprocessor to produce a source deck is shown in Figure 4 on page 92. According to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine (CITYSUB) or a function (CITYFUN).

The DSNAME used for SYSPUNCH specifies a source program library on which the preprocessor output will be placed. Normally compilation would continue and the preprocessor output would be compiled.

```
//OPT4#8  JOB
//STEP2 EXEC  IBMZC,PARM.PLI='MACRO,MDECK,NOCOMPILE,NOSYNTAX'
//PLI.SYSPUNCH DD  DSNAME=HPU8.NEWLIB(FUN),DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN  DD *
 /* GIVEN ZIP CODE, FINDS CITY                              */
 %DCL USE CHAR;
 %USE = 'FUN'                  /* FOR SUBROUTINE, %USE = 'SUB' */ ;
 %IF USE = 'FUN' %THEN %DO;
 CITYFUN: PROC(ZIPIN) RETURNS(CHAR(16)) REORDER; /* FUNCTION        */
                   %END;
              %ELSE %DO;
 CITYSUB: PROC(ZIPIN, CITYOUT) REORDER;     /* SUBROUTINE           */
   DCL CITYOUT CHAR(16);          /* CITY NAME                      */
                   %END;
   DCL (LBOUND, HBOUND) BUILTIN;
   DCL ZIPIN PIC '99999';         /* ZIP CODE                       */
   DCL 1 ZIP_CITY(7) STATIC,     /* ZIP CODE - CITY NAME TABLE      */
         2 ZIP PIC '99999' INIT(
                   95141, 95014, 95030,
                   95051, 95070, 95008,
                   0),            /* WILL NOT LOOK AT LAST ONE     */
         2 CITY CHAR(16) INIT(
                   'SAN JOSE', 'CUPERTINO', 'LOS GATOS',
                   'SANTA CLARA', 'SARATOGA', 'CAMPBELL',
                   'UNKNOWN CITY'); /* WILL NOT LOOK AT LAST ONE  */
   DCL I FIXED BIN(31);
   DO I = LBOUND(ZIP,1) TO              /* SEARCH FOR ZIP IN TABLE    */
         HBOUND(ZIP,1)-1               /* DON'T LOOK AT LAST ELEMENT */
         WHILE(ZIPIN ¬= ZIP(I));
   END;
 %IF USE = 'FUN' %THEN %DO;
   RETURN(CITY(I));                     /* RETURN CITY NAME          */
                   %END;
              %ELSE %DO;
   CITYOUT=CITY(I);                     /* RETURN CITY NAME          */
                   %END;
 END;
```

Figure 4. Using the macro preprocessor to produce a source deck

SQL preprocessor

In general, the coding for your PL/I program will be the same whether or not you
want it to access a DB2 database. However, to retrieve, update, insert, and delete
DB2 data and use other DB2 services, you must use SQL statements. You can use
dynamic and static EXEC SQL statements in PL/I applications.

To communicate with DB2, you need to do the following:
- Code any SQL statements you need, delimiting them with **EXEC SQL**
- Use the DB2 precompiler or, if using DB2 for z/OS Version 7 Release 1 or later.
  compile with the PL/I PP(SQL()) compiler option

Before you can take advantage of EXEC SQL support, you must have authority to
access a DB2 system. Contact your local DB2 Database Administrator for your
authorization.

Note: The PL/I SQL Preprocessor currently does not support DBCS.

Programming and compilation considerations

When you use the PL/I SQL Preprocessor, the PL/I compiler handles your source program containing embedded SQL statements at compile time, without your having to use a separate precompile step. Although the use of a separate precompile step continues to be supported, use of the PL/I SQL Preprocessor is recommended. Interactive debugging with Debug Tool is enhanced when you use the PL/I SQL Preprocessor because you see only the SQL statements while debugging (and not the generated PL/I source). However, you must have DB2 for z/OS Version 7 Release 1 or later to use the SQL preprocessor.

In addition, using the PL/I SQL Preprocessor lifts some of the DB2 precompiler's restrictions on SQL programs. When you process SQL statements with the PL/I SQL Preprocessor, you can now

- use fully-qualified names for structured host variables
- include SQL statements at any level of a nested PL/I program, instead of in only the top-level source file
- use nested SQL INCLUDE statements

Compiling with the PL/I SQL Preprocessor option generates a DB2 database request module (DBRM) along with the usual PL/I compiler outputs such as object module and listing. As input to the DB2 bind process, the DBRM data set contains information about the SQL statements and host variables in the program. Not all of the information in the DBRM is important in terms of the bind or runtime processing, however. For example, if the HOST value in the DBRM specifies a language other than PL/I, there is no reason to be concerned. All this means is that the other language is selected as the installation default for the HOST value, which does not effect the bind or runtime processing of your program.

The PL/I compiler listing includes the error diagnostics (such as syntax errors in the SQL statements) that the PL/I SQL Preprocessor generates.

To use the PL/I SQL Preprocessor, you need to do the following things:

- Specify the following option when you compile your program

      PP(SQL('*options*'))

  This compiler option indicates that you want the compiler to invoke the integrated PL/I SQL preprocessor. Specify a list of SQL processing options in the parenthesis after the SQL keyword. The options can be separated by a comma or by a space and the list of options must be enclosed in quotes (single or double, as long as they match).

  For example, `PP(SQL('DATE(USA),TIME(USA)')` tells the preprocessor to use the USA format for both DATE and TIME data types.

  In addition, for LOB support you must specify the option

      LIMITS(FIXEDBIN(31,63) FIXEDDEC(31))

- Include DD statements for the following data sets in the JCL for your compile step:

  – DB2 load library (*prefix*.SDSNLOAD)

    The PL/I SQL preprocessor calls DB2 modules to do the SQL statement processing. You therefore need to include the name of the DB2 load library data set in the STEPLIB concatenation for the compile step.

  – Library for SQL INCLUDE statements

If your program contains SQL INCLUDE *member-name* statements that specify secondary input to the source program, you need to include the name of the data set that contains *member-name* in the SYSLIB concatenation for the compile step.

– DBRM library

The compilation of the PL/I program generates a DB2 database request module (DBRM) and the DBRMLIB DD statement is required to designate the data set to which the DBRM is written.

– For example, you might have the following lines in your JCL:

```
//STEPLIB  DD  DSN=DSN710.SDSNLOAD,DISP=SHR
//SYSLIB   DD  DSN=PAYROLL.MONTHLY.INCLUDE,DISP=SHR
//DBRMLIB  DD  DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
```

SQL preprocessor options

When you specify SQL preprocessor options, the list of options must be enclosed in quotes (single or double, as long as they match). For example, to specify the DATE(ISO) option, you must specify PP(SQL('DATE(ISO)')).

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

```
                  ┌─────────────────────────────┐
►──┬─────────────────────────────────────────┬──►
   └─SQLFLAG─┬─────────────────────────┬──(──────────────────)──┘
            ├─STD─┬────────────────────────┬─┤
            │     └─(ssname─┬──────────┬──)─┘ │
            │               └─,qualifier─┘    │
            └─IBM───────────────────────────────┘


       ┌─NO──┐            ┌──────┐                ┌─DB2─┐
►──┬──────────────────┬──┬─TIME──(─┼─ISO───┼──)─┬──┬─SQL──(─┼─────┼──)─┬──►
   └─STDSQL──(─┼─────┼──)─┘         ├─USA───┤    │         └─ALL─┘    │
              └─YES─┘               ├─EUR───┤
                                    ├─JIS───┤
                                    └─LOCAL─┘


                              ┌─,──┐
►──┬────────────────────────┬──────────)──)───────────────────────►◄
   └─VERSION──(─┬───────┬──)─┘
               ├─aaaa─┤
               └─AUTO─┘
```
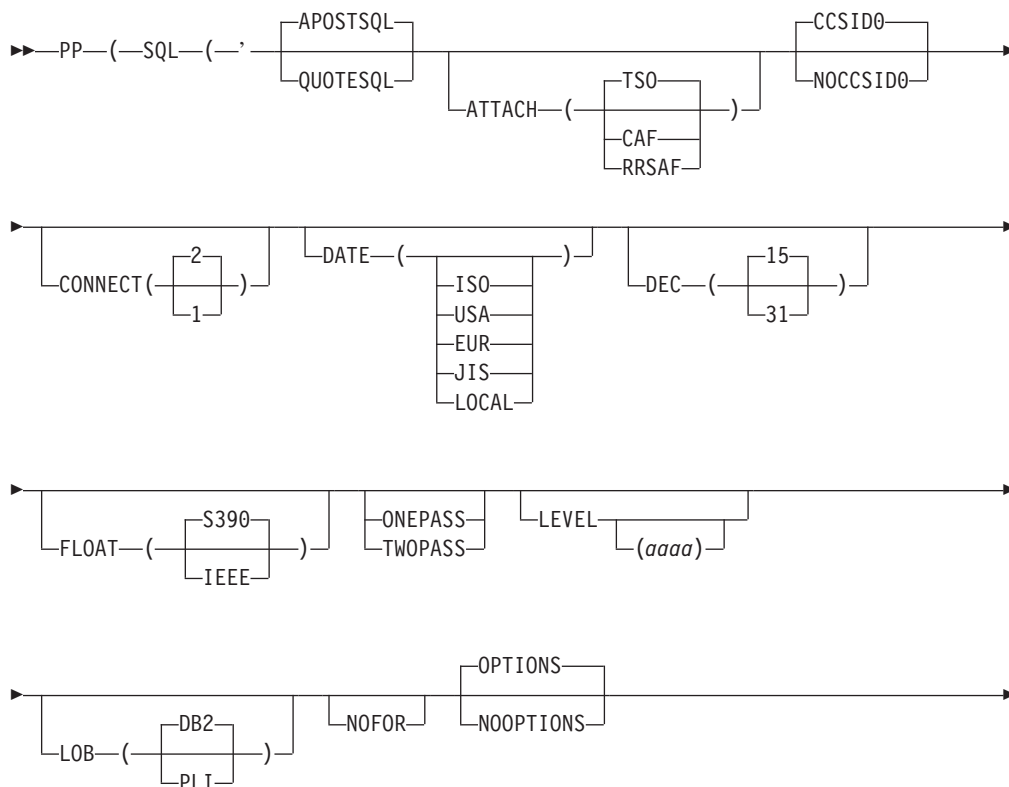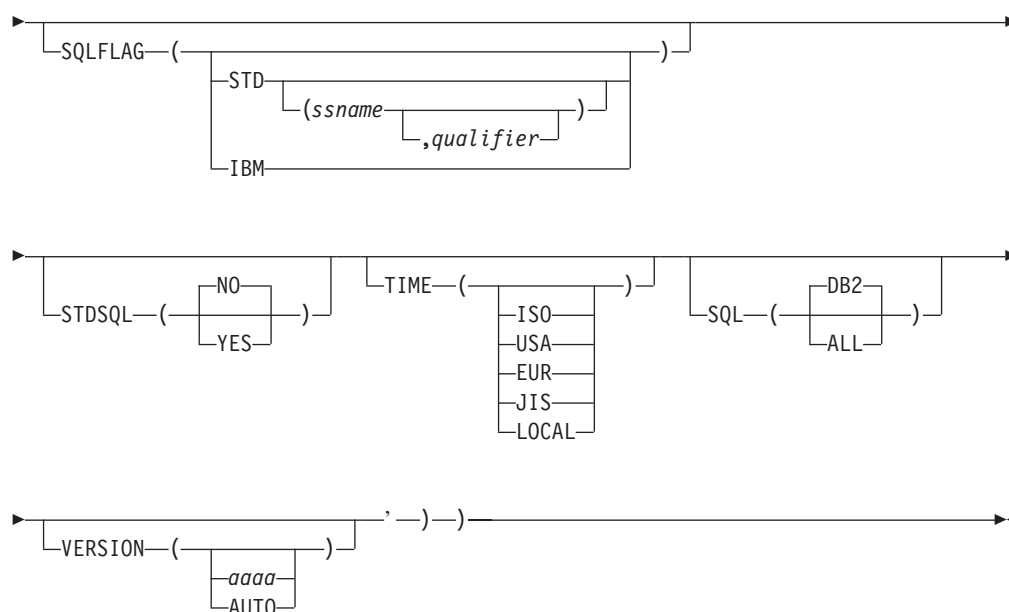
In addition to these PL/I SQL preprocessor options, you may pass DB2
Coprocessor options in on the *PP(SQL('options'))* compiler option. For more
information about DB2 Coprocessor options please consult the *DB2 UDB for z/OS
Application Programming and SQL Guide.*

The table uses a vertical bar(|) to separate mutually exclusive options, and
brackets ([ ]) to indicate that you can sometimes omit the enclosed option.

APOSTSQL

Recognizes the apostrophe (') as the string delimiter and the quotation mark (″)
as the SQL escape character *within SQL statements.*

For compatibility with older PL/I programs which used the DB2 precompiler,
APOSTSQL should be chosen.

APOSTSQL and QUOTESQL are mutually exclusive options.

The default setting is APOSTSQL.

ATTACH(TSO|CAF|RRSAF)

Specifies the attachment facility that the application uses to access DB2. TSO,
CAF and RRSAF. Applications that load the attachment facility can use this
option to specify the correct attachment facility, instead of coding a dummy
DSNHLI entry point.

The default is ATTACH(TSO).

CCSID0

CCSID0 specifies that no host variables be assigned a CCSID value.

If your program updates FOR BIT DATA columns with a data type that is not
BIT data, you will want to choose CCSID0. CCSID0 tells DB2 that the host
variable is not associated with a CCSID, allowing the assignment to be made.
Otherwise, if a host variable that is associated with a CCSID that is not BIT
data is assigned to a FOR BIT DATA column, a DB2 error occurs.

For compatibility with older PL/I programs which used the DB2 precompiler,
CCSID0 should be chosen.

CCSID0 and NOCCSID0 are mutually exclusive options.

The default setting is CCSID0.

CONNECT(2|1)
Determines whether to apply type 1 or type 2 CONNECT statement rules.
- CONNECT(2) Apply rules for the CONNECT (Type 2) statement.
- CONNECT(1) Apply rules for the CONNECT (Type 1) statement.

The default is CONNECT(2).

For more information about this option, refer to the *DB2 SQL Reference* manual.

The CONNECT option can be abbreviated to **CT**.

DATE(ISO|USA|EUR|JIS|LOCAL)
Specifies that date output should always be returned in a particular format, regardless of the format specified as the location default. For a description of these formats, refer to the *DB2 SQL Reference* manual.

The default is in the field DATE FORMAT on the Application Programming Defaults Panel 2 when DB2 is installed.

You cannot use the LOCAL option unless you have a date exit routine.

DEC(15|31)
Specifies the maximum precision for decimal arithmetic operations.

The default is in the field DECIMAL ARITHMETIC on the Application Programming Defaults Panel 1 when DB2 is installed.

FLOAT(S390|IEEE)
Determines whether the contents of floating point host variables are in System/390 hexadecimal format or in IEEE format. An error message is issued if this FLOAT option is different than the PL/I compiler's DEFAULT(HEXADEC|IEEE) option.

The default setting is FLOAT(S390).

GRAPHIC
Indicates that the source code might use mixed data, and that X'0E' and X'0F' are special control characters (shift-out and shift-in) for EBCDIC data.

GRAPHIC and NOGRAPHIC are mutually exclusive options. The default is in the field MIXED DATA on the Application Programming Defaults Panel 1 when DB2 is installed.

LEVEL[(*aaaa*)]
Defines the level of a module, where *aaaa* is any alphanumeric value of up to seven characters. This option is not recommended for general use, and the DSNH CLIST and the DB2I panels do not support it.

You can omit the suboption (*aaaa*). The resulting consistency token is blank.

The LEVEL option can be abbreviated to **L**.

LOB (DB2 | PLI)
Determines the format of the LOB (Large Object) DECLARE and DEFINE statements generated by the SQL preprocessor.

Under LOB(DB2), the generated LOB DECLARE statements are consistent with the form generated by the DB2 Precompiler. Beginning with Enterprise PL/I V3R7 the code generated for all SQL TYPE declarations, including

LOCATOR, ROWID, and *LOB_FILE types, will also be consistent with the DB2 Precompiler output. Choose this option if you are moving from the DB2 Precompiler.

For example, under this option the statement:

```
Dcl BLOB_VAR1 Sql Type Is BLOB(32000);
```

will be converted to:

```
  DCL
 /*$*$*$
 Sql Type Is BLOB(32000)
 $*$*$*/
      1 BLOB_VAR1,
          3 BLOB_VAR1_LENGTH FIXED BIN(31),
          3 BLOB_VAR1_DATA CHAR(32000);
```

Under LOB(PLI), the generated LOB DEFINE statements are consistent with the form generated by the workstation PL/I compilers. Choose this option if you are using PL/I on both the mainframe and workstation platforms to provide cross platform consistency. For example, under this option the statement:

```
Dcl BLOB_VAR1 Sql Type Is BLOB(32000);
```

will be converted to:

```
DEFINE STRUCTURE
   1 BLOB$$x,
      2 BLOB_VAR1_LENGTH  FIXED BIN(31),
      2 BLOB_VAR1_DATA,
         3 BLOB_VAR1_DATA1(1) CHAR(32000);
 DCL BLOB_VAR1   TYPE  BLOB$$x        ;
```

The default is LOB(DB2).

NOCCSID0

NOCCSID0 allows host variables to be assigned a CCSID value.

If your program updates FOR BIT DATA columns with a data type that is not BIT data, you will want to choose CCSID0. CCSID0 tells DB2 that the host variable is not associated with a CCSID, allowing the assignment to be made. Otherwise, if a host variable that is associated with a CCSID that is not BIT data is assigned to a FOR BIT DATA column, a DB2 error occurs.

For compatibility with older PL/I programs which used the DB2 precompiler, CCSID0 should be chosen.

NOCCSID0 and CCSID0 are mutually exclusive options.

The default setting is CCSID0.

NOFOR

In static SQL, NOFOR eliminates the need for the FOR UPDATE of FOR UPDATE OF clause in DECLARE CURSOR statements. When you use NOFOR, your program can make positioned updates to any columns that the program has DB2 authority to update.

When you do not use NOFOR, if you want to make positioned updates to any columns that the program has DB2 authority to update, you need to specify FOR UPDATE with no column list in your DECLARE CURSOR statements. The FOR UPDATE clause with no column list applies to static or dynamic SQL statements.

Whether you use or do not use NOFOR, you can specify FOR UPDATE OF with a column list to restrict updates to only the columns named in the clause and specify the acquisition of update locks.

You imply NOFOR when you use the option STDSQL(YES).

If the resulting DBRM is very large, you might need extra storage when you specify NOFOR or use the FOR UPDATE clause with no column list.

NOGRAPHIC
Indicates the use of X'0E' and X'0F' in a string, but not as control characters.

GRAPHIC and NOGRAPHIC are mutually exclusive options. The default is in the field MIXED DATA on the Application Programming Defaults Panel 1 when DB2 is installed.

NOOPTIONS
Suppresses the SQL Preprocessor options listing.

The NOOPTIONS option can be abbreviated to **NOOPTN**.

ONEPASS
Processes in one pass, to avoid the additional processing time for making two passes. Declarations must appear before SQL references if the ONEPASS option is used.

ONEPASS and TWOPASS are mutually exclusive options.

The default is ONEPASS.

The ONEPASS option can be abbreviated to **ON**.

OPTIONS
Lists SQL Preprocessor options.

The default is OPTIONS.

The OPTIONS option can be abbreviated to **OPTN**.

QUOTESQL
Recognizes the quotation mark (″) as the string delimiter and the apostrophe (') as the SQL escape character *within SQL statements.*

For compatibility with older PL/I programs which used the DB2 precompiler, APOSTSQL should be chosen.

QUOTESQL and APOSTSQL are mutually exclusive options.

The default setting is APOSTSQL.

SQL(ALL | DB2)
Indicates whether the source contains SQL statements other than those recognized by DB2 for z/OS.

SQL(ALL) is recommended for application programs whose SQL statements must execute on a server other than DB2 for z/OS using DRDA access. SQL(ALL) indicates that the SQL statements in the program are not necessarily for DB2 for and z/OS. Accordingly, the SQL statement processor then accepts statements that do not conform to the DB2 syntax rules. The SQL statement processor interprets and processes SQL statements according to distributed relational database architecture (DRDA) rules. The SQL statement processor also issues an informational message if the program attempts to use an IBM SQL reserved words as ordinary identifiers. SQL(ALL) does not affect the limits of the SQL statement processor.

SQL(DB2), the default, means to interpret SQL statements and check syntax for use by DB2 for z/OS. SQL(DB2) is recommended when the database server is DB2 for z/OS.

SQLFLAG(IBM | STD[(*ssname*[,*qualifier*])])**

Specifies the standard used to check the syntax of SQL statements. When statements deviate from the standard, the SQL statement processor writes informational messages (flags) to the output listing. The SQLFLAG option is independent of other SQL statement processor options, including SQL and STDSQL.

IBM checks SQL statements against the syntax of IBM SQL Version 1.

STD checks SQL statements against the syntax of the entry level of the ANSI/ISO SQL standard of 1992. You can also use 86 for option, as in releases before Version 7.

ssname requests semantics checking, using the specified DB2 subsystem name for catalog access. If you do not specify *ssname*, the SQL statement processor checks only the syntax.

qualifier specifies the qualifier used for flagging. If you specify a *qualifier*, you must always specify the *ssname* first. If *qualifier* is *not* specified, the default is the authorization ID of the process that started the SQL statement processor.

STDSQL(NO | YES)

Indicates to which rules the output statements should conform.

STDSQL(YES) indicates that the precompiled SQL statements in the source program conform to certain rules of the SQL standard. STDSQL(NO) indicates conformance to DB2 rules.

The default is in the field STD SQL LANGUAGE on the Application Programming Defaults Panel 2 when DB2 is installed.

STDSQL(YES) automatically implies the NOFOR option.

TIME(ISO | USA | EUR | JIS | LOCAL)

Specifies that time output should always be returned in a particular format, regardless of the format specified as the location default. For a description of these formats, refer to the *DB2 SQL Reference* manual.

The default is in the field TIME FORMAT on the Application Programming Defaults Panel 2 when DB2 is installed.

You cannot use the LOCAL option unless you have a date exit routine.

TWOPASS

Processes in two passes, so that declarations need not precede references.

ONEPASS and TWOPASS are mutually exclusive options.

The default is ONEPASS.

The TWOPASS option can be abbreviated to **TW**.

VERSION(*aaaa* | **AUTO)**

Defines the version identifier of a package, program, and the resulting DBRM. When you specify VERSION, the SQL statement processor creates a version identifier in the program and DBRM. This affects the size of the load module and DBRM. DB2 uses the version identifier when you bind the DBRM to a plan or package.

If you do not specify a version at precompile time, then an empty string is the default version identifier. If you specify AUTO, the SQL statement processor

uses the consistency token to generate the version identifier. If the consistency token is a timestamp, the timestamp is converted into ISO character format and used as the version identifier. The timestamp used is based on the System/370 Store Clock value.

When you compile your PL/I program against a DB2 V9 (or later) database the options provided in the listing are divided into the following two categories:

SQL Preprocessor Options Used
> A list of the PL/I SQL preprocessor options that were in effect at the time of the compile.

DB2 for z/OS Coprocessor Options used
> A list of the DB2 for z/OS Coprocessor options that were in effect at the time of the compile. Please refer to the *DB2 UDB for z/OS Application Programming and SQL Guide* for information on how they are determined.

Coding SQL statements in PL/I applications

You can code SQL statements in your PL/I applications using the language defined in *DB2 UDB for z/OS SQL Reference*. 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 either an SQLCODE variable (if the STDSQL(86) preprocessor option is used) or an SQL communications area (SQLCA). As shown in Figure 5 on page 101, part of an SQLCA consists of an SQLCODE variable and an SQLSTATE variable.

- The SQLCODE value is set by the Database Services after each SQL statement is executed. An application can check the SQLCODE value to determine whether the last SQL statement was successful.
- The SQLSTATE variable can be used as an alternative to the SQLCODE variable when analyzing the result of an SQL statement. Like the SQLCODE variable, the SQLSTATE variable is set by the Database Services after each SQL statement is executed.

The SQLCA declaration should be included by using the EXEC SQL INCLUDE statement:

```
exec sql include sqlca;
```

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

```
Dcl
  1 Sqlca,
    2 sqlcaid       char(8),              /* Eyecatcher = 'SQLCA   '  */
    2 sqlcabc       fixed binary(31),     /* SQLCA size in bytes = 136 */
    2 sqlcode       fixed binary(31),     /* SQL return code          */
    2 sqlerrmc      char(70) var,         /* Error message tokens     */
    2 sqlerrp       char(8),              /* Diagnostic information   */
    2 sqlerrd(0:5)  fixed binary(31),     /* Diagnostic information   */
    2 sqlwarn,                            /* Warning flags            */
      3 sqlwarn0    char(1),
      3 sqlwarn1    char(1),
      3 sqlwarn2    char(1),
      3 sqlwarn3    char(1),
      3 sqlwarn4    char(1),
      3 sqlwarn5    char(1),
      3 sqlwarn6    char(1),
      3 sqlwarn7    char(1),
    2 sqlext,
      3 sqlwarn8    char(1),
      3 sqlwarn9    char(1),
      3 sqlwarna    char(1),
      3 sqlstate    char(5);      /* State corresponding to SQLCODE */
```

Figure 5. 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 EXEC SQL INCLUDE statement:

```
exec sql include sqlda;
```

The SQLDA must not be defined within an SQL declare section.

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

Dcl
  1 Sqlda2 based(Sqldaptr),
    2 sqldaid2     char(8),          /* Eye catcher = 'SQLDA   ' */
    2 sqldabc2     fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
    2 sqln2        fixed binary(15), /* Number of SQLVAR elements*/
    2 sqld2        fixed binary(15), /* # of used SQLVAR elements*/
    2 sqlvar2(Sqlsize refer(sqln2)), /* Variable Description      */
      3 sqlbiglen,
        4 sqllongl fixed binary(31),
        4 sqlrsvdl fixed binary(31),
      3 sqldatal   pointer,
      3 sqltname   char(30) var;

dcl Sqlsize    fixed binary(15);    /* number of sqlvars (sqln) */
dcl Sqldaptr   pointer;
dcl Sqltripled char(1) initial('3');
dcl Sqldoubled char(1) initial('2');
dcl Sqlsingled char(1) initial(' ');
```

Figure 6. The PL/I declaration of an SQL descriptor area

Embedding SQL statements

The first statement of your 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 DSN8710.DEPT
        set   Mgrno = :Mgr_Num
        where Deptno = :Int_Dept;
```

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

SQL style comments ('--') are supported when embedded in SQL statements
beginning with Enterprise PL/I V3R6.

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*) compiler option.

Names: Any valid PL/I variable name can be used for a host variable. The length of a host variable name must not exceed the value *n* specified in the LIMITS(NAME(*n*)) compiler option.

Statement labels: With the exception of the END DECLARE SECTION statement, and the INCLUDE text-file-name statement, executable SQL statements, like PL/I statements, can have a label prefix.

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be a label in the PL/I source code and must be within the scope of any SQL statements affected by the WHENEVER statement.

Using host variables

All host variables used in SQL statements must be explicitly declared. If the ONEPASS option is in effect, a host variable used in an SQL statement must be declared prior to its first use 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.

```
  ┌─────────────────────────────────────────────┐   ;  ──────────────────────────▶◀
  └─Alignment and/or Scope and/or Storage─┘
```
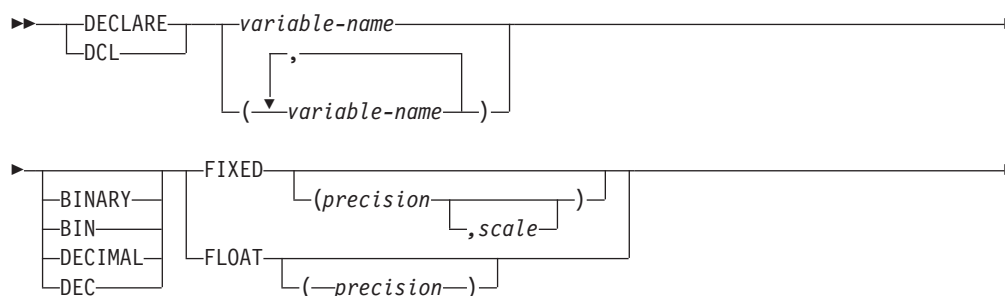
Notes

- BINARY/DECIMAL and FIXED/FLOAT can be specified in either order.
- The precision and scale attributes can also follow BINARY/DECIMAL.
- A value for *scale* can be specified only for DECIMAL FIXED.
- Refer to Table 7 on page 106 for more detailed information.

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

```
▶▶─┬─DECLARE─┬──┬─variable-name────────────┬──────────────────────────────▶
   └─DCL─────┘  │         ┌─,──────────┐    │
               └─(─▼─variable-name─┴─)─┘

▶──┬─CHARACTER─┬──────────────┬─────────────────────────────────────────────▶
   └─CHAR──────┘  └─(length)─┘  ┌─VARYING─┐
                               └─VAR─────┘

▶──┬─────────────────────────────────────────────┬──;──────────────────────▶◀
   └─Alignment and/or Scope and/or Storage─┘
```
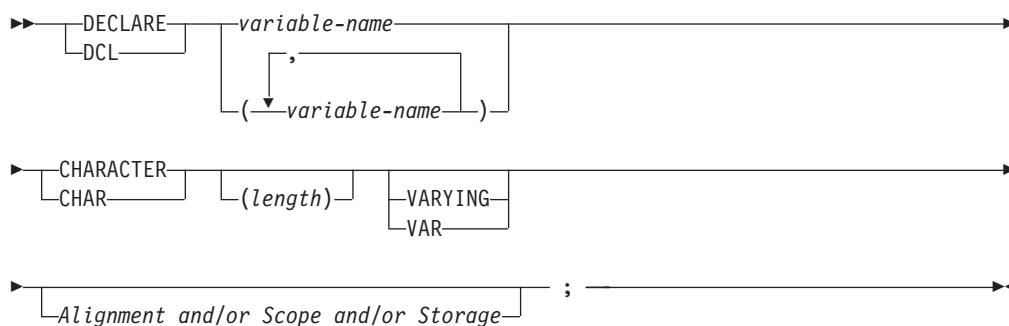
Notes

- 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.

```
▶▶─┬─DECLARE─┬──┬─variable-name────────────┬──GRAPHIC──┬──────────┬──┬─VARYING─┬──▶
   └─DCL─────┘  │         ┌─,──────────┐    │          └─(length)─┘  └─VAR─────┘
               └─(─▼─variable-name─┴─)─┘

▶──┬─────────────────────────────────────────────┬──;──────────────────────▶◀
   └─Alignment and/or Scope and/or Storage─┘
```
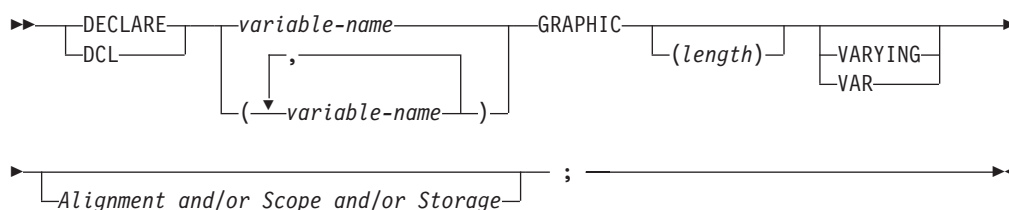
Notes

- 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.

Result set locators: The following figure shows the syntax for valid result set locator declarations.

```
▶▶─┬─DECLARE─┬──┬─variable-name────────────┬──────────────────────────────▶
   └─DCL─────┘  │         ┌─,──────────┐    │
               └─(─▼─variable-name─┴─)─┘
```

►►─SQL TYPE IS RESULT_SET_LOCATOR──────────────────────────────────►

        ├─VARYING─┤
        └─VAR────┘

►─┬──┬──── ; ──────────────►◄
  └─*Alignment and/or Scope and/or Storage*─┘
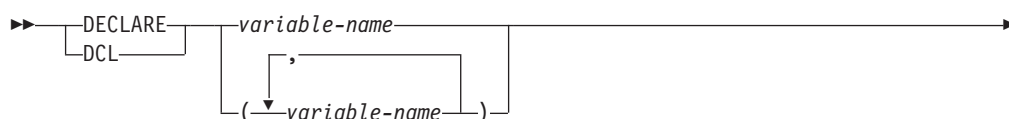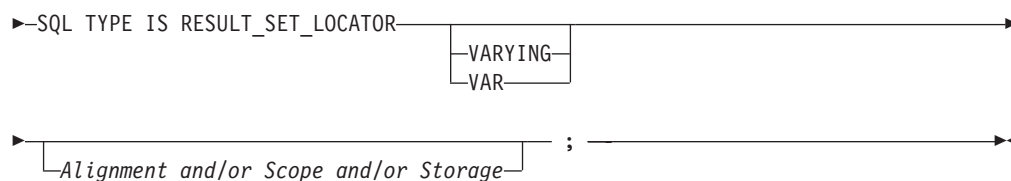
Table locators: The following figure shows the syntax for valid table locators.

►►─┬─DECLARE─┬──┬─*variable-name*──────────────────┬──────────────►
   └─DCL─────┘ │   ┌─,─────────────┐   │
               │   ▼   │   │
               └─(────*variable-name*───)─┘

►─SQL TYPE IS TABLE LIKE table-name AS LOCATOR─ ; ──────────────────►◄
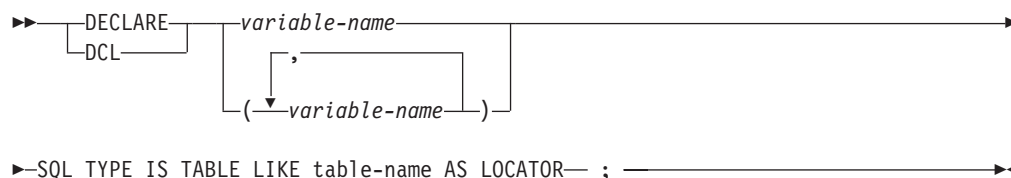
LOB Variables and Locators: The following figure shows the syntax for declarations of BLOB, CLOB, and DBCLOB host variables and locators.

►►─┬─Declare─┬──*PL/I host identifier*─SQL TYPE IS─────────────────►
   └─Dcl─────┘

►─┤ PL/I LOB type ├──┤ PL/I LOB type: ├─────────────────────────────►

►─┬─┬─Binary Large Object─┬──────┬─(─*length*─┬───┬─)──┬────────────►◄
  │ └─BLOB────────────────┘   │   ├─K─┤   │
  │ ┬─Character Large Object─┬   │   ├─M─┤   │
  │ ├─Char Large Object──────┤   │   └─G─┘   │
  │ └─CLOB──────────────────┘   │   │
  │ └─DBCLOB─────────────────────┘   │
  ├─BLOB_LOCATOR───────────────────────────────────────┘
  ├─CLOB_LOCATOR──────
  ├─DBCLOB_LOCATOR────
  ├─BLOB_FILE─────────
  ├─CLOB_FILE─────────
  └─DBCLOB_FILE───────
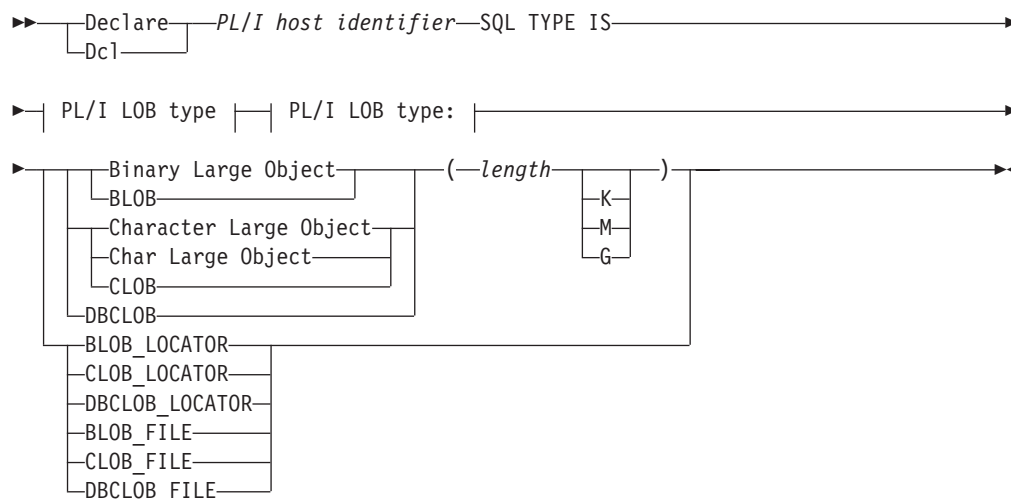
XML File References and LOB Variables: The following figure shows the syntax for declarations of the new ″XML AS″ file reference and LOB variable types.

►►─┬─Declare─┬──*PL/I host identifier*─SQL TYPE IS XML AS───────────►
   └─Dcl─────┘

►─┤ PL/I LOB type ├──┤ PL/I LOB type: ├─────────────────────────────►

►─┬─┬─Binary Large Object─┬─────┬─(─*length*─┬───┬─)──┬─────────────►◄
  │ └─BLOB────────────────┘   │   ├─K─┤   │
  │ ┬─Character Large Object─┬ │   ├─M─┤   │
  │ ├─Char Large Object──────┤ │   └─G─┘   │
  │ └─CLOB──────────────────┘   │   │
  │ └─DBCLOB────────────────────┘   │
  ├─BLOB_FILE──┘
  ├─CLOB_FILE─────────
  └─DBCLOB_FILE───────
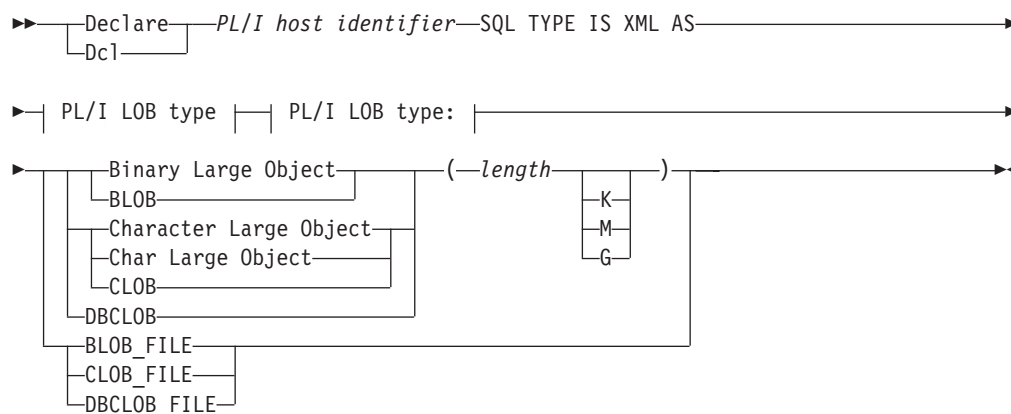
The following constant declarations are generated by the SQL preprocessor and can be used to set the file option variable when you use the file reference host variables:

```
DCL SQL_FILE_READ      FIXED BIN(31) VALUE(2);
DCL SQL_FILE_CREATE    FIXED BIN(31) VALUE(8);
DCL SQL_FILE_OVERWRITE FIXED BIN(31) VALUE(16);
DCL SQL_FILE_APPEND    FIXED BIN(31) VALUE(32);
```

ROWIDs: The following figure shows the syntax for valid declarations of ROWID variables.
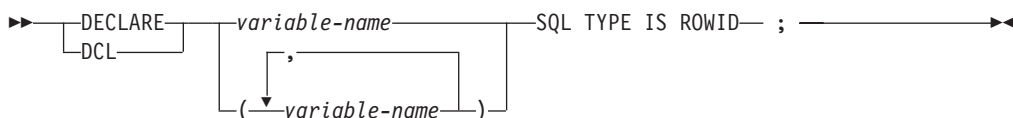
Determining equivalent SQL and PL/I data types

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
BIN FIXED(n), n ranges from 32 to 63	492	8	BIGINT
DEC FIXED(p,s) 0<=p<=15 and 0<=s<=p	484	p (byte 1) s (byte 2)	DECIMAL(p,s)
BIN FLOAT(p), $1 \leq p \leq 21$	480	4	REAL or FLOAT(n) 1<=n<=21
BIN FLOAT(p), $22 \leq p \leq 53$	480	8	DOUBLE PRECISION or FLOAT(n) 22<=n<=53
DEC FLOAT(m), $1 \leq m \leq 6$	480	4	FLOAT (single precision)
DEC FLOAT(m), $7 \leq m \leq 16$	480	8	FLOAT (double precision)
CHAR(n),	452	n	CHAR(n)
CHAR(n) VARYING, $1 \leq n \leq 255$	448	n	VARCHAR(n)
CHAR(n) VARYING, n > 255	456	n	VARCHAR(n)
GRAPHIC(n), $1 \leq n \leq 127$	468	n	GRAPHIC(n)
GRAPHIC(n) VARYING, $1 \leq n \leq 2000$	464	n	VARGRAPHIC(n)
GRAPHIC(n) VARYING, n > 2000	472	n	LONG VARGRAPHIC

Table 8. SQL data types generated from Meta PL/I declarations

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
SQL TYPE IS BLOB(*n*) $1<n<2147483647$	404	*n*	BLOB(*n*)

Table 8. SQL data types generated from Meta PL/I declarations (continued)

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
SQL TYPE IS CLOB(n) 1<n<2147483647	408	n	CLOB(n)
SQL TYPE IS DBCLOB(n) 1<n<1073741823 (2)	412	n	DBCLOB(n) (2)
SQL TYPE IS ROWID	904	40	ROWID
SQL TYPE IS VARBINARY(n) 1<n<32704	908	n	VARBINARY(n)
SQL TYPE IS BINARY(n) 1<n<255	912	n	BINARY(n)
SQL TYPE IS BLOB_FILE	916	267	BLOB File Reference (1)
SQL TYPE IS CLOB_FILE	920	267	CLOB File Reference (1)
SQL TYPE IS DBCLOB_FILE	924	267	DBCLOB File Reference (1)
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB Locator (1)
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB Locator (1)
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB Locator (1)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result Set Locator
SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	976	4	Table Locator (1)

Note:
1. Do not use this data type as a column type.
2. *n* is the number of double-byte characters.

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

Table 9. SQL data types mapped to PL/I declarations

SQL Data Type	PL/I Equivalent	Notes
SMALLINT	BIN FIXED(15)	
INTEGER	BIN FIXED(31)	
BIGINT	BIN FIXED(63)	
DECIMAL(p,s)	DEC FIXED(p) or DEC FIXED(p,s)	p = precision and s = scale; 1 <= p <= 31 and 0 <= s <= p
REAL or FLOAT(n)	BIN FLOAT(p) or DEC FLOAT(m)	$1 \le n \le 21$, $1 \le p \le 21$ and $1 \le m \le 6$
DOUBLE PRECISION, DOUBLE, or FLOAT(n)	BIN FLOAT(p) or DEC FLOAT(m)	$22 \le n \le 53$, $22 \le p \le 53$ and $7 \le m \le 16$
CHAR(n)	CHAR(n)	$1 \le n \le 32767$
VARCHAR(n)	CHAR(n) VAR	
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; $1 \le n \le 2000$

Table 9. SQL data types mapped to PL/I declarations (continued)

SQL Data Type	PL/I Equivalent	Notes
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

Table 10. SQL data types mapped to Meta PL/I declarations

SQL Data Type	PL/I Equivalent	Notes
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.
Table locator	SQL TYPE IS TABLE LIKE *table-name* AS LOCATOR	Use this data type only in a user-defined function or stored procedure to receive rows of a transition table. Do not use this data type as a column type.
BLOB locator	SQL TYPE IS BLOB_LOCATOR	Use this data type only to manipulate data in BLOB columns. Do not use this data type as a column type.
CLOB locator	SQL TYPE IS CLOB_LOCATOR	Use this data type only to manipulate data in CLOB columns. Do not use this data type as a column type.
DBCLOB locator	SQL TYPE IS DBCLOB_LOCATOR	Use this data type only to manipulate data in DBCLOB columns. Do not use this data type as a column type.
BLOB file reference	SQL TYPE IS BLOB_FILE	Use this data type only as a reference to a BLOB file. Do not use this data type as a column type.
CLOB file reference	SQL TYPE IS CLOB_FILE	Use this data type only as a reference to a CLOB file. Do not use this data type as a column type.
DBCLOB file reference	SQL TYPE IS DBCLOB_FILE	Use this data type only as a reference to a DBCLOB file. Do not use this data type as a column type.
BLOB(*n*)	SQL TYPE IS BLOB(*n*)	1<*n*<2147483647
CLOB(*n*)	SQL TYPE IS CLOB(*n*)	1<*n*<2147483647
DBCLOB(*n*)	SQL TYPE IS DBCLOB(*n*)	*n* is the number of double-byte characters. 1<*n*<1073741823
ROWID	SQL TYPE IS ROWID	
XML AS	SQL TYPE IS XML AS ...	Used to describe an XML version of a BLOB, CLOB, DBCLOB, BLOB_FILE, CLOB_FILE, or DBCLOB_FILE

Additional Information on Large Object (LOB) support

General information on LOBs
LOBS, CLOBS, and BLOBS can be as large as 2,147,483,647 bytes long (2 Gigabytes). Double Byte CLOBS can be 1,073,741,823 characters long (1 Gigabyte).

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 one of these structures, depending on the setting of the LOB() precompiler option. When LOB(DB2) is specified and the LOB size is 32767 bytes or less, the generated structure will look like this:

```
  DCL
 /*$*$*$
 SQL TYPE IS lob-type-name (length)
 $*$*$*/
      1 my-identifier-name,
            3 my-identifier-name_LENGTH FIXED BIN(31),
            3 my-identifier-name_DATA CHAR(size1);
```

When LOB(DB2) is specified and the size is greater than 32767 bytes, the generated structure will look like this:

```
  DCL
 /*$*$*$
 SQL TYPE IS lob-type-name (length)
 $*$*$*/
      1 my-identifier-name,
            3 my-identifier-name_LENGTH FIXED BIN(31),
            3 my-identifier-name_DATA,
             4 my-identifier-name_DATA1(size1) CHAR(32767),
             4 my-identifier-name_DATA2 CHAR(size2);
```

When LOB(PLI) is specified and the LOB size is 32767 bytes or less, the generated structure will look like this:

```
  DEFINE STRUCTURE
     1  lob-type$$x,
       2  my-identifier-name_LENGTH FIXED BIN(31),
       2  my-identifier-name_DATA,
         3 my-identifier-name_DATA1 CHAR(size1);
 DCL my-identifier-name TYPE lob-type$$x;
```

When LOB(PLI) is specified and the LOB size is greater than 32767 bytes, the generated structure will look like this:

```
  DEFINE STRUCTURE
     1  lob-type$$x,
       2  my-identifier-name_LENGTH FIXED BIN(31),
       2  my-identifier-name_DATA,
         3 my-identifier-name_DATA1(size1) CHAR(32767),
         3 my-identifier-name_DATA2 CHAR(size2),
  DCL my-identifier-name TYPE lob-type$$x;
```

In these structures, `my-identifier-name` is the name of your PL/I host identifier and `lob-type$$x` is a name generated by the preprocessor. *size1* is an integer value that is the truncated value of *length/32767*. *size2* is the remainder of *length/32767*.

For DBCLOB data types, the generated structure is a little different. Since double-byte characters are two bytes each, *size1* is an integer value that is the truncated value of *length/16383* and *size2* is the remainder of *length/16383*.

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_LOCATOR;
```

The SQL preprocessor would transform the declare into one of these structures,
depending on the setting of the LOB() precompiler option. When LOB(DB2) is
specified the generated code will look like this:

```
   DCL
 /*$*$*$
 SQL TYPE IS lob-type_LOCATOR
 $*$*$*/
  my-identifier-name FIXED BIN(31);
```

When LOB(PLI) is specified the SQL preprocessor would transform this declare
into the following code:

```
 DEFINE ALIAS lob-type_LOCATOR FIXED BIN(31);

 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.

PL/I variable declarations for LOB Support

The following examples provide sample PL/I variable declarations and their
corresponding transformations for LOB support. All of the examples are compiled
with the default LOB(DB2) precompiler option.

Example 1:

```
 DCL my_blob SQL TYPE IS BLOB(2000);
```

After transform:

```
   DCL
 /*$*$*$
 SQL TYPE IS BLOB(2000)
 $*$*$*/
       1 MY_BLOB,
             3 MY_BLOB_LENGTH FIXED BIN(31),
             3 MY_BLOB_DATA CHAR(2000);
```

Example 2:

```
 DCL my_dbclob SQL TYPE IS DBCLOB(4000K);
```

After transform:

```
   DCL
 /*$*$*$
 SQL TYPE IS DBCLOB(4000K)
 $*$*$*/
       1 MY_DBCLOB,
             3 MY_DBCLOB_LENGTH FIXED BIN(31),
             3 MY_DBCLOB_DATA,
               4 MY_DBCLOB_DATA1(250) GRAPHIC(16383),
               4 MY_DBCLOB_DATA2 GRAPHIC(250);
```

Example 3:

```
 DCL my_clob_locator SQL TYPE IS CLOB_LOCATOR;
```

After transform:

```
   DEFINE ALIAS CLOB_LOCATOR FIXED BIN(31);
   DCL my_clob_locator TYPE CLOB_LOCATOR;
```

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.
- 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.

```
►►─┤ Attributes: ├──┬──────BINARY──┬──┬─FIXED──────────────────────────────┬──►◄
                    │  ├──BIN─────┤  │        └─(─precision──┬────────┬─)─┘
                    │  ├─DECIMAL─┤  │                        └─,scale─┘
                    │  └─DEC─────┘  └─FLOAT─────────────────────┐
                    │                      └─(─precision─)─┘
                    └──┬─CHARACTER─┬──┬──────────────┬──┬─VARYING─┐
                       └─CHAR──────┘  └─(─integer─)─┘  └─VAR─────┘
```
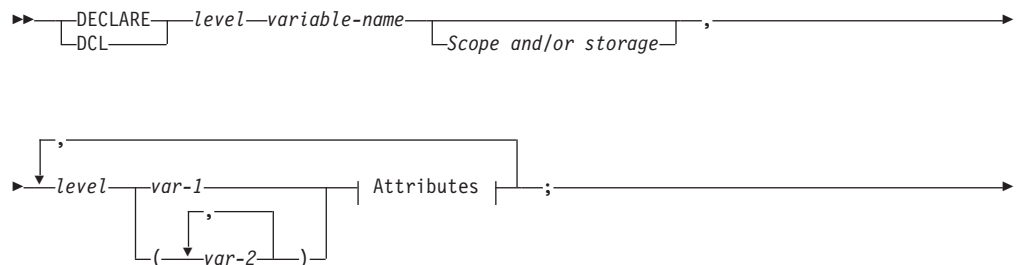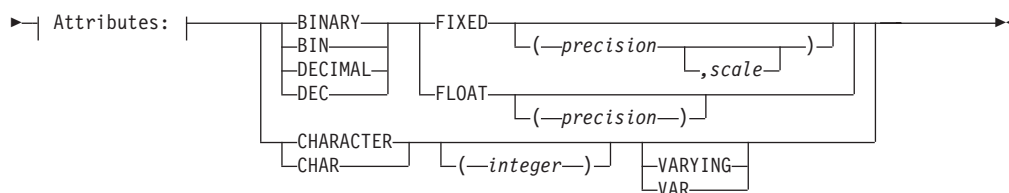
Using indicator variables

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. If an indicator variable array is used, all of the indicator variables in the array must be specified sequentially beginning with the first member. Any out of sequence indicator variable will be flagged as an error.

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.

```
►►──┬─DECLARE─┬──variable-name──┬─BINARY─┬──FIXED(15)─;──────────────────────►◄
    └─DCL─────┘                 └─BIN────┘
```

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

```
►►──┬─DECLARE─┬──┬──variable-name──(─dimension─)──────────────────┬──┬─BINARY─┬──►
    └─DCL─────┘  │                      ┌─,──────────────────┐    │  └─BIN────┘
                 └─(──▼─variable-name──(─dimension─)─┴──)─────┘

►─FIXED(15)─;───────────────────────────────────────────────────────────────►◄
```
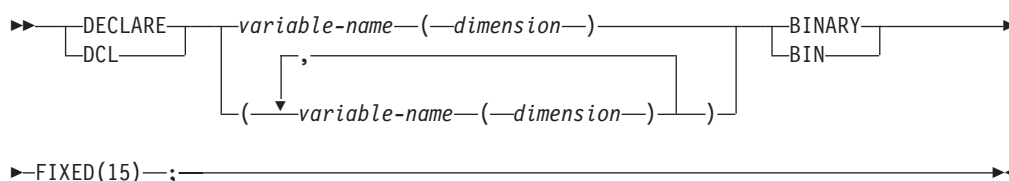
Host structure example

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;
```

Notice that in the first example all of the indicator variables in the array are specified sequentially beginning with the first member. Any out of sequence indicator variable would be flagged as an error.

Using the SQL preprocessor with the compiler user exit (IBMUEXIT)

You can use the IBM supplied compiler user exit (IBMUEXIT) to suppress a message or to change the severity of a message. Because the SQL preprocess handles messages for both itself and the DB2 coprocessor, you need to be aware of the following information.

The SQL preprocessor generates messages in the range IBM7021 - IBM7999. A small subset of these messages, IBM7040, IBM7041, IBM7042, IBM7043 and IBM7044, are 'special' messages that relay message information that was returned from the DB2 coprocessor. These 'special' messages contain two message numbers; the SQL preprocessor assigned IBM704x number (x according to the severity of the DB2 message), and the DB2 message with the DB2 message text. You can not use the user exit to change or suppress this small subset of special SQL preprocessor messages because each of them is used for many separate and individual DB2 coprocessor messages. However, you can use the compiler user exit to change or suppress any of the other 'regular' SQL preprocessor messages.

Here are some examples of how you would change the severity of these two types of SQL preprocessor messages.

Let's consider a 'regular' SQL preprocessor message such as IBM7028 first:

```
IBM7028I W Reference var-name is ambiguous.
```

If you wished to change the severity of a 'regular' SQL preprocessor message like IBM7028 from 4 (warning) to 12 (severe) your modified SYSUEXIT DD card would look like this:

```
 Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-----------------------
 'SQL'     7028      12          0       Ambiguous reference
```

For more information on the fields and values of this example you can refer to Chapter 19, "Using user exits," on page 385.

Next, let's consider one of the 'special' SQL preprocessor messages, IBM7041:

```
IBM7041I W    DSNH527I DSNHOPTS  THE PRECOMPILER ATTEMPTED TO USE
              THE DB2-SUPPLIED DSNHDECP MODULE
```

Notice that it contains two message numbers: 'IBM7041' is assigned by the SQL preprocessor and 'DSNH527' is assigned by the DB2 coprocessor. If you wished to change the severity of a 'special' SQL preprocessor message like IBM7021 from 4 (warning) to 12 (severe) your modified SYSUEXIT DD card would look like this:

```
 Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-----------------------
 'SQL'      527       12          0      DB2 coprocessor message
```

Notice that the Facility ID is the same for both kinds of SQL preprocessor messages. However, for the 'special' SQL preprocessor message you would use the message number provided by the DB2 coprocessor, in this case '527', as the message number you want to modify.

DECLARE STATEMENT statements

The preprocessor ignores all DECLARE STATEMENT statements.

CICS Preprocessor

You can use EXEC CICS statements in PL/I applications that run as transactions under CICS.

If you do not specify the PP(CICS) option, EXEC CICS statements are parsed and variable references in them are validated. If they are correct, no messages are issued as long as the NOCOMPILE option is in effect. If you do not invoke the CICS translator and the COMPILE option is in effect, the compiler will issue S-level messages.

The compiler will invoke the CICS preprocessor if you specify the CICS suboption of the PP option. For compatibility, it will also invoke the CICS preprocessor if the compiler sees any of these options: CICS, XOPT or XOPTS. However, you should not specify one of these options and also the PP(CICS) options.

Programming and compilation considerations

When you are developing programs for execution under CICS, all the EXEC CICS commands must be translated in one of two ways:

- by the command language translator provided by CICS in a job step prior to the PL/I compilation
- by the PL/I CICS preprocessor as part of the PL/I compilation (this requires CICS TS 2.2 or later)

To use the CICS preprocessor, you must also specify the PP(CICS) compile-time option.

Unless you specify CICS as one of the suboptions of the PP(CICS) option, the compiler will flag any EXEC CICS statements in the source. Similarly, it will flag any EXEC CPSM or EXEC DLI statements if you do not specify CPSM or DLI respectively as a suboption of the PP(CICS) option.

If your CICS program is a MAIN procedure, you must also compile it with the SYSTEM(CICS) or the SYSTEM(MVS) option. If compiled with SYSTEM(MVS), the

PTFs for runtime APAR PQ91318 must be applied. NOEXECOPS is implied with this option and all parameters passed to the MAIN procedure must be POINTERs. For a description of the SYSTEM compile-time option, see "SYSTEM" on page 64.

If your want your CICS program to be reentrant and if your program uses FILEs or CONTROLLED variables, then you must compile it with the NOWRITABLE as well.

If your CICS program includes any files or uses any macros that contain EXEC CICS statements, you must also run the MACRO preprocessor before your code is translated (in either of the ways described above). If you are using the CICS preprocessor, you can specify this with one PP option as illustrated in the following example:

```
pp (macro(...)  cics(...)  )
```

Finally, in order to use the CICS preprocessor, you must have the CICS SDFHLOAD dataset as part of the STEPLIB DD for the PL/I compiler.

CICS preprocessor options

There are many options supported by CICS translator. For a description of these options, see the *CICS Application Programming Guide*.

Note that these options should be enclosed in quotes (single or double, as long as they match). For instance, to invoke the CICS preprocessor with the EDF option, you should specify the option PP(CICS('EDF')).

Coding CICS statements in PL/I applications

You can code CICS statements in your PL/I applications using the language defined in *CICS on Open Systems Application Programming Guide*. Specific requirements for your CICS code are described in the sections that follow.

Embedding CICS statements
If you use the CICS translator, rather than the integrated preprocessor, then 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

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 must observe the following restrictions on PL/I CICS programs:
- Macro-level CICS is not supported.
- PL/I input or output cannot be used except for:
  - PUT FILE(SYSPRINT)
  - CALL PLIDUMP
- The PLISRTx built-in subroutines cannot be used.
- Routines written in a language other than PL/I cannot be called from a PL/I CICS program if those routines contain their own EXEC CICS statements. If you want to communicate with a non-PL/I program that contains EXEC CICS statements, you must use EXEC CICS LINK or EXEC CICS XCTL to do so.

Although PUT FILE(SYSPRINT) is permitted under CICS, you should generally not use it in production programs as it will degrade performance.

Since the CICS EIB address is only generated by either the CICS translator or the PL/I CICS preprocessor for an OPTIONS(MAIN) program, it is the user's responsibility to establish the addressability to the EIB for the OPTIONS(FETCHABLE) routine.

Addressability can be achieved either by using this command:
```
EXEC CICS ADDRESS EIB(DFHEIPTR)
```

or by passing the EIB address as an argument to the CALL statement that invokes the external procedure.

Error-handling

Language Environment prohibits the use of the following EXEC CICS commands in any PL/I ON-unit or in any code called from a PL/I ON-unit.
- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

All other EXEC CICS commands are allowed within an ON-unit. However, they must be coded using the NOHANDLE option, the RESP option, or the RESP2 option.

Chapter 3. Using PL/I cataloged procedures

This chapter describes the standard cataloged procedures supplied by IBM for use with the IBM Enterprise PL/I for z/OS compiler. It explains how to invoke them, and how to temporarily or permanently modify them. The Language Environment SCEERUN data set must be located in STEPLIB and accessible to the compiler when you use any of the cataloged procedures described in this chapter.

A cataloged procedure is a set of job control statements, stored in a library, that includes one or more EXEC statements, each of which can be followed by one or more DD statements. You can retrieve the statements by naming the cataloged procedure in the PROC parameter of an EXEC statement in the input stream.

You can use cataloged procedures to save time and reduce Job Control Language (JCL) errors. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job. You should review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for your own conventions.

IBM-supplied cataloged procedures

The PL/I cataloged procedures supplied for use with Enterprise PL/I for z/OS are:

IBMZC
> Compile only

IBMZCB
> Compile and bind

IBMZCPL
> Compile, prelink, and link-edit

IBMZCBG
> Compile, bind, and run

IBMZCPLG
> Compile, prelink, link-edit, and run

IBMZCPG
> Compile, prelink, load, and run

Cataloged procedures IBMZCB and IBMZCBG use features of the program management binder introduced in DFSMS/MVS® 1.4 in place of the prelinker supplied with Language Environment. These procedures produce a program object in a PDSE.

Cataloged procedures IBMZCPL, IBMZCPLG and IBMZCPG use the prelinker supplied with Language Environment and produce a load module in PDS. Use these procedures if you do not want to use a PDSE. The information in this section describes the procedure steps of the different cataloged procedures. For a description of the individual statements for compiling and link editing, see"Invoking the compiler under z/OS using JCL" on page 135 and *z/OS Language Environment Programming Guide*. These cataloged procedures do not include a DD statement for the input data set; you must always provide one. The example shown in Figure 7 on page 118 illustrates the JCL statements you might use to invoke the cataloged procedure IBMZCBG to compile, bind, and run a PL/I program.

Enterprise PL/I requires a minimum REGION size of 32M. Large programs require more storage. If you do not specify REGION on the EXEC statement that invokes the cataloged procedure you are running, the compiler uses the default REGION size for your site. The default size might or might not be adequate, depending on the size of your PL/I program.

If you compile your programs with optimization turned on, the REGION size (and time) required may be much, much larger.

For an example of specifying REGION on the EXEC statement, see Figure 7.

```
//COLEGO    JOB
//STEP1     EXEC IBMZCBG, REGION.PLI=32M
//PLI.SYSIN DD *
               .
               .
               .
     (insert PL/I program to be compiled here)
               .
               .
               .
 /*
```

Figure 7. Invoking a cataloged procedure

Compile only (IBMZC)

The IBMZC cataloged procedure, shown in Figure 8 on page 119, includes only one procedure step, in which the options specified for the compilation are OBJECT and OPTIONS. (IBMZPLI is the symbolic name of the compiler.) In common with the other cataloged procedures that include a compilation procedure step, IBMZC does not include a DD statement for the input data set; you must always supply an appropriate statement with the qualified ddname PLI.SYSIN.

The OBJECT compile-time option causes the compiler to place the object module, in a syntax suitable for input to the linkage editor, in the standard data set defined by the DD statement with the name SYSLIN. This statement defines a temporary data set named &&LOADSET on a sequential device; if you want to retain the object module after the end of your job, you must substitute a permanent name for &&LOADSET (that is, a name that does not start with &&) and specify KEEP in the appropriate DISP parameter for the last procedure step that used the data set. You can do this by providing your own SYSLIN DD statement, as shown below. The data set name and disposition parameters on this statement will override those on the IBMZC procedure SYSLIN DD statement. In this example, the compile step is the only step in the job.

```
//PLICOMP EXEC IBMZC
//PLI.SYSLIN  DD  DSN=MYPROG,DISP=(MOD,KEEP)
//PLI.SYSIN   DD ...
```

The term MOD in the DISP parameter in Figure 8 on page 119 allows the compiler to place more than one object module in the data set, and PASS ensures that the data set is available to a later procedure step providing a corresponding DD statement is included there.

The SYSLIN SPACE parameter allows an initial allocation of 1 cylinder and, if necessary, 15 further allocations of 1 cylinder (a total of 16 cylinders).

```
//IBMZC    PROC LNGPRFX='IBMZ.V3R7M0',LIBPRFX='CEE',
//              SYSLBLK=3200
//*
//**********************************************************************
//*                                                                    *
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2007                        *
//* ALL RIGHTS RESERVED.                                               *
//*                                                                    *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                       *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                        *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                               *
//*                                                                    *
//**********************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 7 MODIFICATION 0
//*
//*  COMPILE A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE     USAGE
//*   LNGPRFX   IBMZ.V3R7M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE               PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200              BLKSIZE FOR OBJECT DATA SET
//*
//*  USER MUST SUPPLY //PLI.SYSIN DD STATEMENT THAT IDENTIFIES
//*   LOCATION OF COMPILER INPUT
//*
//**********************************************************************
//* COMPILE STEP
//**********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 8. Cataloged Procedure IBMZC

Compile and bind (IBMZCB)

The IBMZCB cataloged procedure, shown in Figure 9 on page 120, includes two procedure steps: PLI, which is identical to cataloged procedure IBMZC, and BIND, which invokes the Program Management binder (symbolic name IEWBLINK) to bind the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement BIND specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 8 (that is, if a severe or unrecoverable error occurs during compilation).

```
//IBMZCB  PROC LNGPRFX='IBMZ.V3R7M0',LIBPRFX='CEE',
//            SYSLBLK=3200,GOPGM=GO
//*
//**********************************************************************
//*                                                                    *
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2007                        *
//* ALL RIGHTS RESERVED.                                               *
//*                                                                    *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                       *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                        *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                               *
//*                                                                    *
//**********************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 7 MODIFICATION 0
//*
//*  COMPILE AND BIND A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V3R7M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//**********************************************************************
//* COMPILE STEP
//**********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//**********************************************************************
//* BIND STEP
//**********************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//             PARM='XREF,COMPAT=PM3'
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
```

Figure 9. Cataloged Procedure IBMZCB

The Program Management binder always places the program objects it creates in
the standard data set defined by the DD statement with the name SYSLMOD. This
statement in the cataloged procedure specifies a new temporary library &&GOSET,
in which the program object will be placed and given the member name GO. In
specifying a temporary library, the cataloged procedure assumes that you will run
the program object in the same job; if you want to retain the program object, you
must substitute your own statement for the DD statement with the name
SYSLMOD.

Compile, bind, and run (IBMZCBG)

The IBMZCBG cataloged procedure, shown in Figure 10, includes three procedure steps: PLI, BIND, and GO. PLI and BIND are identical to the two procedure steps of IBMZCB, and GO runs the program object created in the step BIND. The GO step is executed only if no severe or unrecoverable errors occurred in the preceding procedure steps.

Input data for the compilation procedure step should be specified in a DD statement with the name PLI.SYSIN, and for the GO step in a DD statement with the name GO.SYSIN.

```
//IBMZCBG  PROC LNGPRFX='IBMZ.V3R7M0',LIBPRFX='CEE',
//            SYSLBLK=3200,GOPGM=GO
//*
//**********************************************************************
//*                                                                    *
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2007                        *
//* ALL RIGHTS RESERVED.                                               *
//*                                                                    *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                       *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                        *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                               *
//*                                                                    *
//**********************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 7 MODIFICATION 0
//*
//*  COMPILE, BIND, AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V3R7M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//**********************************************************************
//* COMPILE STEP
//**********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 10. Cataloged Procedure IBMZCBG (Part 1 of 2)

```
//*********************************************************************
//* BIND STEP
//*********************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//             PARM='XREF,COMPAT=PM3'
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
//*********************************************************************
//* RUN STEP
//*********************************************************************
//GO       EXEC PGM=*.BIND.SYSLMOD,COND=((8,LT,PLI),(8,LE,BIND))
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 10. Cataloged Procedure IBMZCBG (Part 2 of 2)

Compile, prelink, and link-edit (IBMZCPL)

The IBMZCPL cataloged procedure, shown in Figure 11 on page 123, includes three
procedure steps: PLI, which is identical to cataloged procedure IBMZC; PLKED,
which invokes the Language Environment prelinker; and LKED, which invokes the
linkage editor (symbolic name IEWL) to link-edit the object module produced in
the first procedure step.

Input data for the compilation procedure step requires the qualified ddname
PLI.SYSIN. The COND parameter in the EXEC statement LKED specifies that this
procedure step should be bypassed if the return code produced by the compiler is
greater than 8 (that is, if a severe or unrecoverable error occurs during
compilation).

```
//IBMZCPL PROC LNGPRFX='IBMZ.V3R7M0',LIBPRFX='CEE',
//           SYSLBLK=3200,PLANG=EDCPMSGE,GOPGM=GO
//*
//**********************************************************************
//*                                                                    *
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2007                        *
//* ALL RIGHTS RESERVED.                                               *
//*                                                                    *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                       *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                        *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                               *
//*                                                                    *
//**********************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 7 MODIFICATION 0
//*
//*  COMPILE, PRELINK, LINK-EDIT A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE     USAGE
//*   LNGPRFX   IBMZ.V3R7M0       PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE               PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200              BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE          PRELINKER MESSAGES MEMBER NAME
//*   GOPGM     GO                MEMBER NAME FOR LOAD MODULE
//*
//**********************************************************************
//* COMPILE STEP
//**********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 11. Cataloged Procedure IBMZCPL (Part 1 of 2)

```
//*********************************************************************
//* PRE-LINK-EDIT STEP
//*********************************************************************
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI)
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSN=&&PLNK,DISP=(,PASS),
//             UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//*********************************************************************
//* LINK-EDIT STEP
//*********************************************************************
//LKED     EXEC PGM=IEWL,PARM='XREF',COND=((8,LT,PLI),(8,LE,PLKED))
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1))
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,SPACE=(1024,(200,20)),
//             DCB=BLKSIZE=1024
//SYSIN    DD  DUMMY
```

Figure 11. Cataloged Procedure IBMZCPL (Part 2 of 2)

The linkage editor always places the load modules it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the load module will be placed and given the member name GO. In specifying a temporary library, the cataloged procedure assumes that you will run the load module in the same job; if you want to retain the module, you must substitute your own statement for the DD statement with the name SYSLMOD.

The SYSLIN DD statement in Figure 11 on page 123 shows how to concatenate a data set defined by a DD statement named SYSIN with the primary input (SYSLIN) to the linkage editor. You could place linkage editor control statements in the input stream by this means, as described in the *z/OS Language Environment Programming Guide.*

Compile, prelink, link-edit, and run (IBMZCPLG)

The IBMZCPLG cataloged procedure, shown in Figure 12 on page 125, includes four procedure steps: PLI, PLKED, LKED, and GO. PLI, PLKED, and LKED are identical to the three procedure steps of IBMZCPL, and GO runs the load module created in the step LKED. The GO step is executed only if no severe or unrecoverable errors occurred in the preceding procedure steps.

Input data for the compilation procedure step should be specified in a DD statement with the name PLI.SYSIN, and for the GO step in a DD statement with the name GO.SYSIN.

```
//IBMZCPLG PROC LNGPRFX='IBMZ.V3R7M0',LIBPRFX='CEE',
//             SYSLBLK=3200,PLANG=EDCPMSGE,GOPGM=GO
//*
//**********************************************************************
//*                                                              *
//* LICENSED MATERIALS - PROPERTY OF IBM                         *
//*                                                              *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2007                  *
//* ALL RIGHTS RESERVED.                                         *
//*                                                              *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                 *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                  *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                         *
//*                                                              *
//**********************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 7 MODIFICATION 0
//*
//*  COMPILE, PRELINK, LINK-EDIT AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V3R7M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MEMBER NAME
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//**********************************************************************
//* COMPILE STEP
//**********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//**********************************************************************
//* PRE-LINK-EDIT STEP
//**********************************************************************
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI)
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSN=&&PLNK,DISP=(,PASS),UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
```

Figure 12. Cataloged Procedure IBMZCPLG (Part 1 of 2)

```
//*********************************************************************
//* LINK-EDIT STEP
//*********************************************************************
//LKED     EXEC PGM=IEWL,PARM='XREF',COND=((8,LT,PLI),(8,LE,PLKED))
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1))
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,SPACE=(1024,(200,20)),
//             DCB=BLKSIZE=1024
//SYSIN    DD  DUMMY
//*********************************************************************
//* RUN STEP
//*********************************************************************
//GO       EXEC PGM=*.LKED.SYSLMOD,
//             COND=((8,LT,PLI),(8,LE,PLKED),(8,LE,LKED))
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 12. Cataloged Procedure IBMZCPLG (Part 2 of 2)

Compile, prelink, load and run (IBMZCPG)

The IBMZCPG cataloged procedure, shown in Figure 13 on page 127, achieves the
same results as IBMZCPLG but uses the loader instead of the linkage editor.
Instead of using four procedure steps (compile, prelink, link-edit, and run), it has
only three (compile, prelink, and load-and-run). The third procedure step runs the
loader program. The loader program processes the object module produced by the
compiler and runs the resultant executable program immediately. You must
provide input data for the compilation step by supplying a qualified ddname
PLI.SYSIN.

The use of the loader imposes certain restrictions on your PL/I program; before
using this cataloged procedure, see *z/OS Language Environment Programming Guide*,
which explains how to use the loader.

```
//IBMZCPG PROC LNGPRFX='IBMZ.V3R7M0',LIBPRFX='CEE',
//             SYSLBLK=3200,PLANG=EDCPMSGE
//*
//**********************************************************************
//*                                                                    *
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2007                        *
//* ALL RIGHTS RESERVED.                                               *
//*                                                                    *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                       *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                        *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                               *
//*                                                                    *
//**********************************************************************
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 7 MODIFICATION 0
//*
//*  COMPILE, PRELINK, LOAD AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V3R7M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MEMBER NAME
//*
//**********************************************************************
//* COMPILE STEP
//**********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB  DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//**********************************************************************
//* PRE-LINK-EDIT STEP
//**********************************************************************
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI)
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSN=&&PLNK,DISP=(,PASS),
//             UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN    DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
```

Figure 13. Cataloged Procedure IBMZCPG (Part 1 of 2)

```
//********************************************************************
//* LOAD AND RUN STEP
//********************************************************************
//GO       EXEC PGM=LOADER,PARM='MAP,PRINT',
//             COND=((8,LT,PLI),(8,LE,PLKED))
//STEPLIB  DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLOUT  DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 13. Cataloged Procedure IBMZCPG (Part 2 of 2)

For more information on other cataloged procedures, see *z/OS Language Environment Programming Guide*.

Invoking a cataloged procedure

To invoke a cataloged procedure, specify its name in the PROC parameter of an EXEC statement. For example, to use the cataloged procedure IBMZC, you could include the following statement in the appropriate position among your other job control statements in the input stream:

```
//stepname EXEC PROC=IBMZC
```

You do not need to code the keyword PROC. If the first operand in the EXEC statement does not begin PGM= or PROC=, the job scheduler interprets it as the name of a cataloged procedure. The following statement is equivalent to that given above:

```
//stepname EXEC IBMZC
```

If you include the parameter MSGLEVEL=1 in your JOB statement, the operating system will include the original EXEC statement in its listing, and will add the statements from the cataloged procedure. In the listing, cataloged procedure statements are identified by XX or X/ as the first two characters; X/ signifies a statement that was modified for the current invocation of the cataloged procedure.

You might be required to modify the statements of a cataloged procedure for the duration of the job step in which it is invoked, either by adding DD statements or by overriding one or more parameters in the EXEC or DD statements. For example, cataloged procedures that invoke the compiler require the addition of a DD statement with the name SYSIN to define the data set containing the source statements. Also, whenever you use more than one standard link-edit procedure step in a job, you must modify all but the first cataloged procedure that you invoke if you want to run more than one of the load modules.

Specifying multiple invocations

You can invoke different cataloged procedures, or invoke the same cataloged procedure several times, in the same job. No special problems are likely to arise unless more than one of these cataloged procedures involves a link-edit procedure step, in which case you must take the following precautions to ensure that all your load modules can be run.

When the linkage editor creates a load module, it places the load module in the standard data set defined by the DD statement with the name SYSLMOD. When the binder creates a program object, it places the program object in the PDSE defined by the DD statement with the name SYSLMOD. In the absence of a linkage editor NAME statement, the linkage editor or the binder uses the member name specified in the DSNAME parameter as the name of the module. In the standard cataloged procedures, the DD statement with the name SYSLMOD always specifies a temporary library &&GOSET with the member name GO.

If you use the cataloged procedure IBMZCBG twice within the same job to compile, bind, and run two PL/I programs, and do not name each of the two program objects that the binder creates, the first program object runs twice, and the second one not at all.

To prevent this, use one of the following methods:
- Delete the library &&GOSET at the end of the GO step. In the first invocation of the cataloged procedure at the end of the GO step, add a DD statement with the syntax:

  ```
  //GO.SYSLMOD DD DSN=&&GOSET,
  //   DISP=(OLD,DELETE)
  ```
- Modify the DD statement with the name SYSLMOD in the second and subsequent invocations of the cataloged procedure so as to vary the names of the load modules. For example:

  ```
  //BIND.SYSLMOD DD DSN=&&GOSET(GO1)
  ```

  and so on.
- Use the NAME linkage editor option to give a different name to each program object and change each job step EXEC statement to specify the running of the program object with the name for that job step.

To assign a membername to the program object, you can use the linkage editor NAME option with the DSNAME parameter on the SYSLMOD DD statement. When you use this procedure, the membername **must** be identical to the name on the NAME option if the EXEC statement that runs the program refers to the SYSLMOD DD statement for the name of the module to be run.

Another option is to give each program a different name by using GOPGM on the EXEC procedure statement. For example:

```
// EXEC IBMZCBG,GOPGM=GO2
```

Modifying the PL/I cataloged procedures

You can modify a cataloged procedure temporarily by including parameters in the EXEC statement that invokes the cataloged procedure, or by placing additional DD statements after the EXEC statement. Temporary modifications apply only for the duration of the job step in which the procedure is invoked. They do not affect the master copy of the cataloged procedure in the procedure library.

Temporary modifications can apply to EXEC or DD statements in a cataloged procedure. To change a parameter of an EXEC statement, you must include a corresponding parameter in the EXEC statement that invokes the cataloged procedure. To change one or more parameters of a DD statement, you must include a corresponding DD statement after the EXEC statement that invokes the cataloged procedure. Although you cannot add a new EXEC statement to a cataloged procedure, you can always include additional DD statements.

EXEC statement

If a parameter of an EXEC statement that invokes a cataloged procedure has an unqualified name, the parameter applies to all the EXEC statements in the cataloged procedure. The effect on the cataloged procedure depends on the parameters, as follows:

- PARM applies to the first procedure step and nullifies any other PARM parameters.
- COND and ACCT apply to all the procedure steps.
- TIME and REGION apply to all the procedure steps and override existing values.

For example, the statement:

```
//stepname EXEC IBMZCBG,PARM='OFFSET',REGION=32M
```

- Invokes the cataloged procedure IBMZCBG.
- Substitutes the option OFFSET for OBJECT and OPTIONS in the EXEC statement for procedure step PLI.
- Nullifies the PARM parameter in the EXEC statement for procedure step BIND.
- Specifies a region size of 32M for all three procedure steps.

To change the value of a parameter in only one EXEC statement of a cataloged procedure, or to add a new parameter to one EXEC statement, you must identify the EXEC statement by qualifying the name of the parameter with the name of the procedure step. For example, to alter the region size for procedure step PLI only in the preceding example, code:

```
//stepname EXEC PROC=IBMZCBG,PARM='OFFSET',REGION.PLI=90M
```

A new parameter specified in the invoking EXEC statement overrides completely the corresponding parameter in the procedure EXEC statement.

You can nullify all the options specified by a parameter by coding the keyword and equal sign without a value. For example, to suppress the bulk of the linkage editor listing when invoking the cataloged procedure IBMZCBG, code:

```
//stepname EXEC IBMZCBG,PARM.BIND=
```

DD statement

To add a DD statement to a cataloged procedure, or to modify one or more parameters of an existing DD statement, you must include a DD statement with the form `procstepname.ddname` in the appropriate position in the input stream. If `ddname` is the name of a DD statement already present in the procedure step identified by `procstepname`, the parameters in the new DD statement override the corresponding parameters in the existing DD statement; otherwise, the new DD statement is added to the procedure step. For example, the statement:

```
//PLI.SYSIN DD *
```

adds a DD statement to the procedure step PLI of cataloged procedure IBMZC and the effect of the statement:

```
//PLI.SYSPRINT DD SYSOUT=C
```

is to modify the existing DD statement SYSPRINT (causing the compiler listing to be transmitted to the system output device of class C).

Overriding DD statements must appear after the procedure invocation and in the same order as they appear in the cataloged procedure. Additional DD statements can appear after the overriding DD statements are specified for that step.

To override a parameter of a DD statement, code either a revised form of the parameter or a replacement parameter that performs a similar function (for example, SPLIT for SPACE). To nullify a parameter, code the keyword and equal sign without a value. You can override DCB subparameters by coding only those you wish to modify; that is, the DCB parameter in an overriding DD statement does not necessarily override the entire DCB parameter of the corresponding statement in the cataloged procedures.

Chapter 4. Compiling your program

This chapter describes how to invoke the compiler under z/OS UNIX System Services (z/OS UNIX) and the job control statements used for compiling under z/OS. The Language Environment SCEERUN data set must be accessible to the compiler when you compile your program.

Invoking the compiler under z/OS UNIX

To compile your program under the z/OS UNIX environment, use the **pli** command.

```
►►──pli─┬─────────────────────────────────────┬──────────────►◄
        │  ┌─────────────────┐ ┌────────────┐ │
        └──▼─command_line_option─┴─▼─input_file─┴─┘
```
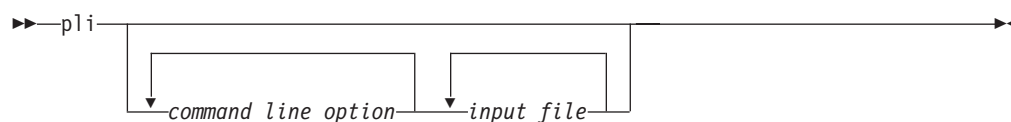
command_line_option
> You can specify a `command_line_option` in the following ways:
> - **-q**option
> - Option flag (usually a single letter preceded by -)
>
> If you choose to specify compile-time options on the command line, the format differs from either setting them in your source file using %PROCESS statements. See "Specifying compile-time options under z/OS UNIX" on page 134.

input_file
> The z/OS UNIX 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.

Input files

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

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

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

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

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

Specifying compile-time options under z/OS UNIX

Enterprise PL/I provides compile-time options to change any of the 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.

Refer to "Compile-time option descriptions" on page 3 for a description of these
options.

When you specify options on the command line, they override the default settings
of the option. They are overridden by options set in the source file.

You can specify compile-time options on the command line in three ways:
- -**q***option_keyword* (compiler-specific)
- Single and multiletter flags
- -q+/u/myopts.txt

-qoption_keyword

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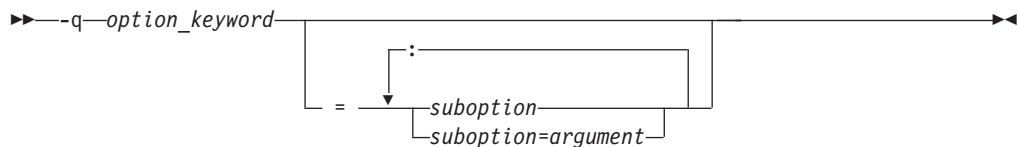
You can have multiple -**q***options* on the same command line, but they must be
separated by blanks. Option keywords can appear in either uppercase or
lowercase, but you must specify the -**q** in lowercase.

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 54). To specify RULES(LAXDCL) on the command line, you would enter:

-qrules=ibm**:**laxdcl

The LIMITS option ("LIMITS" on page 36) is slightly more complex since each of
its suboptions also has an argument. You would specify
LIMITS(EXTNAME(31),FIXEDDEC(15)) on the command line as shown in the
following example:

-qlimits=extname=31:fixeddec=15

Single and multiletter flags

The z/OS UNIX family of compilers uses a number of common conventional flags.
Each language has its own set of additional flags.

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

```
pli samp.pli -I/home/test3/include
```

In this case, `/home/test3/include` is an include directory to be searched for INCLUDE files.

Each flag option should be specified as a separate argument.

Table 11. Compile-time option flags supported by Enterprise PL/I under z/OS UNIX

Option	Description
-c	Compile only.
-e	Create names and entries for a fetchable load module.
-I<*dir*>*	Add path <*dir*> to the directories to be searched for INCLUDE files. -I must be followed by a path and only a single path is allowed per -I option. To add multiple paths, use multiple -I options. There 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*>.
-v	Display compile and link steps and execute them.
-#	Display compile and link steps, but do not execute them.

Note: *You must specify an argument where indicated; otherwise, the results are unpredictable.

Invoking the compiler under z/OS using JCL

Although you will probably use cataloged procedures rather than supply all the JCL required for a job step that invokes the compiler, you should be familiar with these statements so that you can make the best use of the compiler and, if necessary, override the statements of the cataloged procedures.

So-called ″batch compilation″, whereby one compilation produced more than one object deck, is not supported.

The following section describes the JCL needed for compilation. The IBM-supplied cataloged procedures described in "IBM-supplied cataloged procedures" on page 117 contain these statements. You need to code them yourself only if you are not using the cataloged procedures.

EXEC statement

The basic EXEC statement is:

```
//stepname EXEC PGM
```

512K is required for the REGION parameter of this statement.

If you compile your programs with optimization turned on, the REGION size (and time) required may be much, much larger.

The PARM parameter of the EXEC statement can be used to specify one or more of the optional facilities provided by the compiler. These facilities are described under

"Specifying options in the EXEC statement" on page 138. See Chapter 1, "Using compiler options and facilities," on page 3 for a description of the options.

DD statements for the standard data sets

The compiler requires several standard data sets, the number of data sets depends on the optional facilities specified. You must define these data sets in DD statements with the standard ddnames shown, together with other characteristics of the data sets, in Table 12. The DD statements SYSIN, SYSUT1, and SYSPRINT are always required.

You can store any of the standard data sets on a direct access device, but you must include the SPACE parameter in the DD statement. This parameter defines the data set to specify the amount of auxiliary storage required. The amount of auxiliary storage allocated in the IBM-supplied cataloged procedures should suffice for most applications.

Table 12. Compiler standard data sets

Standard DDNAME	Contents of data set	Possible device classes[1]	Record format (RECFM)	Record size (LRECL)
SYSDEBUG	TEST(SEPARATE) output	SYSDA	F,FB	>=80 and <=1024
SYSDEFSD	XINFO(DEF) output	SYSDA	F,FB	128
SYSIN	Input to the compiler	SYSSQ	F,FB,U VB,V	<101(100) <105(104)
SYSLIB	Source statements for INCLUDE files	SYSDA	F,FB,U V,VB	<101 <105
SYSLIN	Object module	SYSSQ	FB	80
SYSPRINT	Listing, including messages	SYSSQ	VBA	137
SYSPUNCH	Preprocessor output, compiler output	SYSSQ SYSCP	FB	80 or MARGINS() value
SYSUT1	Temporary workfile	SYSDA	F	4051
SYSXMI	XINFO(XMI) output	SYSDA	VB	16383
SYSXMLSD	XINFO(XML) output	SYSDA	VB	16383

Notes:

The only value for compile-time SYSPRINT that can be overridden is BLKSIZE.

1. The possible device classes are:
   SYSSQ   Sequential device
   SYSDA   direct access device

   Block size can be specified except for SYSUT1. The block size and logical record length for SYSUT1 is chosen by the compiler.

Input (SYSIN)

Input to the compiler must be a data set defined by a DD statement with the name SYSIN. This data set must have CONSECUTIVE organization. The input must be one or more external PL/I procedures. If you want to compile more than one external procedure in a single job or job step, precede each procedure, except possibly the first, with a %PROCESS statement.

80-byte records are commonly used as the input medium for PL/I source programs. The input data set can be on a direct access device or some other sequential media. The input data set can contain either fixed-length records (blocked or unblocked), variable-length records (coded or uncoded), or undefined-length records. The maximum record size is 100 bytes.

The maximum number of lines in the input file is 999,999.

When data sets are concatenated for input to the compiler, the concatenated data sets must have similar characteristics (for example, block size and record format).

Output (SYSLIN, SYSPUNCH)

Output in the form of one or more object modules from the compiler will be stored in the data set SYSLIN if you specify the OBJECT compile-time option. This data set is defined by the DD statement.

The object module is always in the form of 80-byte fixed-length records, blocked or unblocked. If the BLKSIZE is specified for SYSLIN and is something other than 80, then the LRECL must be specified as 80.

The SYSLIN DD must name either a temporary dataset or a permanent dataset: it cannot specify a concatenation of datasets of any type.

The SYSLIN DD must specify a sequential dataset, not a PDS or PDSE.

The data set defined by the DD statement with the name SYSPUNCH is also used to store the output from the preprocessor if you specify the MDECK compile-time option.

Temporary workfile (SYSUT1)

The compiler requires a data set for use as a temporary workfile. It is defined by a DD statement with the name SYSUT1, and is known as the *spill file*. It must be on a direct access device, and must not be allocated as a multi-volume data set.

The spill file is used as a logical extension to main storage and is used by the compiler and by the preprocessor to contain text and dictionary information. The LRECL and BLKSIZE for SYSUT1 is chosen by the compiler based on the amount of storage available for spill file pages.

The DD statements given in this publication and in the cataloged procedures for SYSUT1 request a space allocation in blocks of 1024 bytes. This is to insure that adequate secondary allocations of direct access storage space are acquired.

Listing (SYSPRINT)

The compiler generates a listing that includes all the source statements that it processed, information relating to the object module, and, when necessary, messages. Most of the information included in the listing is optional, and you can specify those parts that you require by including the appropriate compile-time options. The information that can appear, and the associated compile-time options, are described under "Using the compiler listing" on page 79.

You must define the data set, in which you wish the compiler to store its listing, in a DD statement with the name SYSPRINT. This data set must have CONSECUTIVE organization. Although the listing is usually printed, it can be

stored on any sequential or direct access device. For printed output, the following statement will suffice if your installation follows the convention that output class A refers to a printer:

```
//SYSPRINT DD SYSOUT=A
```

Source Statement Library (SYSLIB)

If you use the %INCLUDE statement to introduce source statements into the PL/I program from a library, you can either define the library in a DD statement with the name SYSLIB, or you can choose your own ddname (or ddnames) and specify a ddname in each %INCLUDE statement.

The DD should specify a PDS or PDSE, but not the actual member. For example to include the file HEADER from the library SYSLIB using the dataset INCLUDE.PLI, the %INCLUDE statement would like

```
  %INCLUDE HEADER;
```

or

```
  %INCLUDE SYSLIB( HEADER );
```

The DD statement should be

```
SYSLIB  DD DISP=SHR,DSN=INCLUDE.PLI
```

but it should not be

```
SYSLIB  DD DISP=SHR,DSN=INCLUDE.PLI(HEADER)
```

All %INCLUDE files must have the same record format (fixed, variable, undefined), the same logical record length, and the same left and right margins as the SYSIN source file.

The BLOCKSIZE of the library must be less than or equal to 32,760 bytes.

The maximum number of lines in any one include file is 999,999.

Specifying options

For each compilation, the IBM-supplied or installation default for a compile-time option applies unless it is overridden by specifying the option in a %PROCESS statement or in the PARM parameter of an EXEC statement.

An option specified in the PARM parameter overrides the default value, and an option specified in a %PROCESS statement overrides both that specified in the PARM parameter and the default value.

Note: When conflicting attributes are specified either explicitly or implicitly by the specification of other options, the latest implied or explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden in this way.

Specifying options in the EXEC statement

To specify options in the EXEC statement, code PARM= followed by the list of options, in any order separating the options with commas and enclosing the list within single quotation marks, for example:

```
//STEP1 EXEC PGM=IBMZPLI,PARM='OBJECT,LIST'
```

Any option that has quotation marks, for example `MARGINI('c')`, must have the quotation marks duplicated. The length of the option list must not exceed 100 characters, including the separating commas. However, many of the options have an abbreviated syntax that you can use to save space. If you need to continue the statement onto another line, you must enclose the list of options in parentheses (instead of in quotation marks) enclose the options list on each line in quotation marks, and ensure that the last comma on each line except the last line is outside of the quotation marks. An example covering all the above points is as follows:

```
//STEP1 EXEC PGM=IBMZPLI,PARM=('AG,A',
//       'C,F(I)',
// 'M,MI(''X''),NEST,STG,X')
```

If you are using a cataloged procedure, and want to specify options explicitly, you must include the PARM parameter in the EXEC statement that invokes it, qualifying the keyword PARM with the name of the procedure step that invokes the compiler. For example:

```
//STEP1 EXEC nnnnnnn,PARM.PLI='A,LIST'
```

Specifying options in the EXEC statement using an options file

Another way to specify options in the EXEC statement is by declaring all your options in an options file and coding the following:

```
//STEP1 EXEC PGM=IBMZPLI,PARM='+DD:OPTIONS'
```

This method allows you to provide a consistent set of options that you frequently use. This is especially effective if you want other programmers to use a common set of options. It also gets you past the 100-character limit.

The MARGINS option does not apply to options files: the data in column 1 will be read as part of the options. Also, if the file is F-format, any data after column 72 will be ignored.

The parm string can contain ″normal″ options and can point to more than one options file. For instance, to specify the option LIST as well as options from both the file in the GROUP DD and in the PROJECT DD, you could specify

```
  PARM='LIST +DD:GROUP +DD:PROJECT'
```

The options in the PROJECT file would have precedence over options in the GROUP file.

Also, in this case, the LIST option might be turned off by a NOLIST option specified in either of the options files. To insure that the LIST option is on, you could specify

```
  PARM='+DD:GROUP +DD:PROJECT LIST'
```

Options files may also be used under z/OS UNIX. For example, in USS, to compile sample.pli with options from the file /u/pli/group.opt, you would specify

```
    pli -q+/u/pli/group.opt  sample.pli
```

Earlier releases of the compiler used the character '@' as the trigger character that preceded the options file specification. This character is not part of the invariant set of EBCDIC code points, and for that reason the character '+', which is invariant, is preferred. However, the '@' character may be still be used as long as it is specified with the hex value '7C'x.

Specifying compile-time options

Chapter 5. Link-editing and running

After compilation, your program consists of one or more object modules that contain unresolved references to each other, as well as references to the Language Environment run-time library. These references are resolved during link-editing (statically) or during execution (dynamically). There are two ways to link-edit statically:

1. Use the prelinker prior to the traditional link step
2. Link without the prelinker, which is similar to linking with PL/I for MVS & VM except that depending on which compile-time options you use, you may now need to use a PDSE to hold the resultant load module.

After you compile your PL/I program, the next step is to link and run your program with test data to verify that it produces the results you expect. When using Enterprise PL/I we recommend you select the method of linking without the prelinker (as described in Item 2 above).

Language Environment provides the run-time environment and services you need to execute your program. For instructions on linking and running PL/I and all other Language Environment-conforming language programs, refer to *z/OS Language Environment Programming Guide.* For information about migrating your existing PL/I programs to Language Environment, see *Enterprise PL/I for z/OS Compiler and Run-Time Migration Guide.*

Link-edit considerations

If you compile with the option RENT or the option LIMITS(EXTNAME(n)) with n > 8, then you must use the prelinker or use a PDSE for your linker output.

Using the binder

You must place the binder output into a PDSE.

When linking a DLL, you must specify any needed definition side-decks during the bind step.

You can use the binder in place of the prelinker and linkage-editor, with the following exceptions:

- Releases of CICS prior to CICS Transaction Server 1.3 do not support PDSEs. From CICS Transaction Server 1.3 onwards, there is support in CICS for PDSEs. Please refer to the CICS Transaction Server for z/OS Release Guide, GC34-5701 for a list of prerequisite APAR fixes.
- MTF does not support PDSEs.

Using the prelinker

If you use the prelinker, you must prelink together in one job step all object decks defining external references in any of the input object decks.

For instance, if A and B are separately compiled programs and A statically calls B, then you cannot prelink A and B separately and then later link them together. Instead. you must link A and B together in one prelink job.

Using the ENTRY card

If you are building a module that will be fetched and which has an Enterprise PL/I routine as its entry point, then the ENTRY card should specify the name of that PL/I entry point. If the module is to be fetched from Enterprise PL/I, you may specify CEESTART on the ENTRY card, although this is strongly not recommended. However, if the module is to be fetched from COBOL or assembler, then the ENTRY card absolutely must specify the name of the PL/I entry point into the module and not CEESTART.

Run-time considerations

You can specify run-time options as parameters passed to the program initialization routine. You can also specify run-time options in the PLIXOPT variable. It might also prove beneficial, from a performance standpoint, if you alter your existing programs by using the PLIXOPT variable to specify your run-time options and recompiling your programs. For a description of using PLIXOPT, see *z/OS Language Environment Programming Guide*.

To simplify input/output at the terminal, various conventions have been adopted for stream files that are assigned to the terminal. Three areas are affected:
1. Formatting of PRINT files
2. The automatic prompting feature
3. Spacing and punctuation rules for input.

Note: No prompting or other facilities are provided for record I/O at the terminal, so you are strongly advised to use stream I/O for any transmission to or from a terminal.

Formatting conventions for PRINT files

When a PRINT file is assigned to the terminal, it is assumed that it will be read as it is being printed. Spacing is therefore reduced to a minimum to reduce printing time. The following rules apply to the PAGE, SKIP, and ENDPAGE keywords:
- PAGE options or format items result in three lines being skipped.
- SKIP options or format items larger than SKIP (2) result in three lines being skipped. SKIP (2) or less is treated in the usual manner.
- The ENDPAGE condition is never raised.

Changing the format on PRINT files

If you want normal spacing to apply to output from a PRINT file at the terminal, you must supply your own tab table for PL/I. This is done by declaring an external structure called PLITABS in the main program or in a program linked with the main program and initializing the element PAGELENGTH to the number of lines that can fit on your page. This value differs from PAGESIZE, which defines the number of lines you want to print on the page before ENDPAGE is raised (see Figure 15 on page 143). If you require a PAGELENGTH of 64 lines, declare PLITABS as shown in Figure 14 on page 143. For information on overriding the tab table, see "Overriding the tab control table" on page 205.

If your code contains a declare for PLITABS, not only must the pagesize, linesize and other values be valid, but the first field in the PLITABS structure must also be valid. This field is supposed to hold the offset to the field specifying the number of tabs set by the structure, and the Enterprise PL/I library code will not work correctly if this is not true.

```
DCL 1 PLITABS STATIC EXTERNAL,
  ( 2   OFFSET INIT (14),
    2    PAGESIZE INIT (60),
    2    LINESIZE INIT (120),
    2    PAGELENGTH INIT (64),
    2    FILL1 INIT (0),
    2    FILL2 INIT (0),
    2    FILL3 INIT (0),
    2    NUMBER_OF_TABS INIT (5),
    2    TAB1 INIT (25),
    2    TAB2 INIT (49),
    2    TAB3 INIT (73),
    2    TAB4 INIT (97),
    2    TAB5 INIT (121)) FIXED BIN (15,0);
```

Figure 14. Declaration of PLITABS. This declaration gives the standard page size, line size and tabulating positions
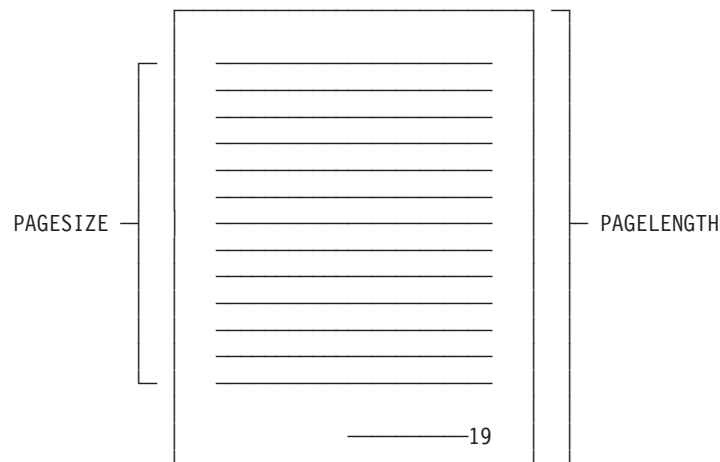

```
PAGELENGTH:  the number of lines that can be printed on a page

PAGESIZE:    the number of lines that will be printed on a page
             before the ENDPAGE condition is raised
```

Figure 15. PAGELENGTH and PAGESIZE. PAGELENGTH defines the size of your paper, PAGESIZE the number of lines in the main printing area.

Automatic prompting

When the program requires input from a file that is associated with a terminal, it issues a prompt. This takes the form of printing a colon on the next line and then skipping to column 1 on the line following the colon. This gives you a full line to enter your input, as follows:

```
:
(space for entry of your data)
```

This type of prompt is referred to as a primary prompt.

Overriding automatic prompting

You can override the primary prompt by making a colon the last item in the
request for the data. You cannot override the secondary prompt. For example, the
two PL/I statements:

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));
```

result in the terminal displaying:

```
ENTER TIME OF PERIHELION
:        (automatic prompt)
(space for entry of data)
```

However, if the first statement has a colon at the end of the output, as follows:

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

the sequence is:

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

Note: The override remains in force for only one prompt. You will be
automatically prompted for the next item unless the automatic prompt is
again overridden.

Punctuating long input lines

Line continuation character

To transmit data that requires 2 or more lines of space at the terminal as one
data-item, type an SBCS hyphen as the last character in each line except the last
line. For example, to transmit the sentence "this data must be transmitted as one
unit." you enter:

```
:'this data must be transmitted -
+:as one unit.'
```

Transmission does not occur until you press ENTER after "unit.'". The hyphen is
removed. The item transmitted is called a "logical line."

Note: To transmit a line whose last data character is a hyphen or a PL/I minus
sign, enter two hyphens at the end of the line, followed by a null line as the
next line. For example:

```
xyz--
(press ENTER only, on this line)
```

Punctuating GET LIST and GET DATA statements

For GET LIST and GET DATA statements, a comma is added to the end of each
logical line transmitted from the terminal, if the programmer omits it. Thus there is
no need to enter blanks or commas to delimit items if they are entered on separate
logical lines. For the PL/I statement GET LIST(A,B,C); you can enter at the
terminal:

```
:1
+:2
+:3
```

This rule also applies when entering character-string data. Therefore, a character
string must transmit as one logical line. Otherwise, commas are placed at the break
points. For example, if you enter:

```
:'COMMAS SHOULD NOT BREAK
+:UP A CLAUSE.'
```

the resulting string is: "COMMAS SHOULD NOT BREAK, UP A CLAUSE." The
comma is not added if a hyphen was used as a line continuation character.

Automatic padding for GET EDIT

For a GET EDIT statement, there is no need to enter blanks at the end of the line.
The data will be padded to the specified length. Thus, for the PL/I statement:

```
GET EDIT (NAME) (A(15));
```

you can enter the 5 characters SMITH. The data will be padded with ten blanks so
that the program receives the fifteen characters:

```
'SMITH          '
```

Note: A single data item must transmit as a logical line. Otherwise, the first line
transmitted will be padded with the necessary blanks and taken as the
complete data item.

Use of SKIP for terminal input: All uses of SKIP for input are interpreted as
SKIP(1) when the file is allocated to the terminal. SKIP(1) is treated as an
instruction to ignore all unused data on the currently available logical line.

ENDFILE

The end-of-file can be entered at the terminal by keying in a logical line that
consists of the two characters "/*". Any further attempts to use the file without
closing it result in the ENDFILE condition being raised.

SYSPRINT considerations

The PL/I standard SYSPRINT file is shared by multiple enclaves within an
application. You can issue I/O requests, for example STREAM PUT, from the same
or different enclaves. These requests are handled using the standard PL/I
SYSPRINT file as a file which is common to the entire application. The SYSPRINT
file is implicitly closed only when the application terminates, not at the termination
of the enclave.

The standard PL/I SYSPRINT file contains user-initiated output only, such as
STREAM PUTs. Run-time library messages and other similar diagnostic output are
directed to the Language Environment MSGFILE. See the *z/OS Language
Environment Programming Guide* for details on redirecting SYSPRINT file output to
the Language Environment MSGFILE.

To be shared by multiple enclaves within an application, the PL/I SYSPRINT file
must be declared as an EXTERNAL FILE constant with a file name of SYSPRINT
and also have the attributes STREAM and OUTPUT as well as the (implied)
attribute of PRINT, when OPENed. This is the standard SYSPRINT file as
defaulted by the compiler.

There exists only one standard PL/I SYSPRINT FILE within an application and
this file is shared by all enclaves within the application. For example, the
SYSPRINT file can be shared by multiple nested enclaves within an application or
by a series of enclaves that are created and terminated within an application by the
Language Environment preinitialization function. To be shared by an enclave
within an application, the PL/I SYSPRINT file must be declared in that enclave.

The standard SYSPRINT file cannot be shared by passing it as a file argument between enclaves. The declared attributes of the standard SYSPRINT file should be the same throughout the application, as with any EXTERNALly declared constant. PL/I does not enforce this rule. Both the TITLE option and the MSGFILE(SYSPRINT) option attempt to route SYSPRINT to another data set. As such, if the two options were used together, there will be a conflict and the TITLE option will be ignored.

Having a common SYSPRINT file within an application can be an advantage to applications that utilize enclaves that are closely tied together. However, since all enclaves in an application write to the same shared data set, this might require some coordination among the enclaves.

The SYSPRINT file is opened (implicitly or explicitly) when first referenced within an enclave of the application. When the SYSPRINT file is CLOSEd, the file resources are released (as though the file had never been opened) and all enclaves are updated to reflect the closed status.

If SYSPRINT is utilized in a multiple enclave application, the LINENO built-in function only returns the current line number until after the first PUT or OPEN in an enclave has been issued. This is required in order to maintain full compatibility with old programs.

The COUNT built-in function is maintained at an enclave level. It always returns a value of zero until the first PUT in the enclave is issued. If a nested child enclave is invoked from a parent enclave, the value of the COUNT built-in function is undefined when the parent enclave regains control from the child enclave.

The TITLE option can be used to associate the standard SYSPRINT file with different operating system data sets, keeping in mind that a particular open association has to be closed before another one is opened. This association is retained across enclaves for the duration of the open.

PL/I condition handling associated with the standard PL/I SYSPRINT file retains its current semantics and scope. For example, an ENDPAGE condition raised within a child enclave will only invoke an established ON-unit within that child enclave. It does not cause invocation of an ON-unit within the parent enclave.

The tabs for the standard PL/I SYSPRINT file can vary when PUTs are done from different enclaves, if the enclaves contain a user PLITABS table.

If the PL/I SYSPRINT file is utilized as a RECORD file or as a STREAM INPUT file, PL/I supports it at an individual enclave or task level, but not as a shareable file among enclaves. If the PL/I SYSPRINT file is open at the same time with different file attributes (e.g. RECORD and STREAM) in different enclaves of the same application, results are unpredictable.

SYSPRINT may also be shared between code compiled by Enterprise PL/I and by older PL/I compilers, but the following must all apply:
- SYSPRINT must be declared as STREAM OUTPUT
- the application must not be running under TSO
- the application must not be running under CICS

Using FETCH in your routines

In Enterprise PL/I, you can fetch routines compiled by PL/I, C, COBOL or assembler.

Fetching Enterprise PL/I routines

Almost all the restrictions imposed by the older PL/I compilers on fetched modules have been removed. So a FETCHed module can now:

- Fetch other modules
- Perform any I/O operations on any PL/I file. The file can be opened either by the fetched module, by the main module, or by some other fetched module.
- ALLOCATE and FREE its own CONTROLLED variables

There are, however, a few restrictions on a Enterprise PL/I module that is to be fetched. These restrictions are:

1. OPTIONS(FETCHABLE) must be specified on the PROCEDURE statement of the routine that provides the entry point into the module to be fetched - or the DLLINIT compiler option must be used to apply OPTIONS(FETCHABLE) to that procedure.
2. The ENTRY card should specify the name of that PL/I entry point
   - If the module is to be fetched from Enterprise PL/I, you may specify CEESTART on the ENTRY card, although this is strongly not recommended.
   - However, if the module is to be fetched from COBOL or assembler, then the ENTRY card absolutely must specify the name of the PL/I entry point into the module and not CEESTART.
3. If the RENT compiler option was used to compile any of the fetched code, then the module must be linked as a DLL.
4. If the NORENT compiler option was used to compile the fetching code, then any fetched module must consist only of NORENT code.
5. If the RENT compiler option was used to compile the fetching code, then the ENTRY that is fetched must not be declared in the fetching module as OPTIONS(COBOL) or OPTIONS(ASM). If you want to avoid passing descriptors in this situation, you should specify the OPTIONS(NODESCRIPTOR) attribute on the ENTRY declare.

As an illustration of these restrictions, consider the compiler user exit. If you specify the EXIT compile-time option, the compiler will fetch and call a Enterprise PL/I module named IBMUEXIT.

First note that the compiler user exit must be compiled with the RENT option since the compiler expects it to be a DLL.

In accordance with Item 1 above, the PROCEDURE statement for this routine looks like:

```
ibmuexit:
  proc ( addr_Userexit_Interface_Block,
         addr_Request_Area )
  options( fetchable  );

  dcl addr_Userexit_Interface_Block  pointer byvalue;

  dcl addr_Request_Area              pointer byvalue;
```

In accordance with Item 3 above, the linker option DYNAM=DLL must be specified when linking the user exit into a DLL. The DLL must be linked either into a PDSE or into a temporary dataset (in which case DSNTYPE=LIBRARY must be specified on the SYSLMOD DD statement).

All the JCL to compile, link, and invoke the user exit is given in the JCL below in Figure 16 on page 149. The one significant difference between the sample below and the code excerpts above is that, in the code below, the fetched user exit does not receive two BYVALUE pointers to structures, but instead it receives the two structures BYADDR. In order to make this change work, the code specifies OPTIONS(NODESCRIPTOR) on each of its PROCEDURE statements.

```
//*
//**********************************************************************
//* compile the user exit
//**********************************************************************
//PLIEXIT EXEC PGM=IBMZPLI,
//              REGION=256K
//STEPLIB   DD DSN=IBMZ.V3R7M0.SIBMZCMP,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//              SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//              SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*Process or('|') not('!');
*Process limits(extname(31));

 /********************************************************************/
 /*                                                                  */
 /*   NAME - IBMUEXIT.PLI                                            */
 /*                                                                  */
 /*   DESCRIPTION                                                    */
 /*      User-exit sample program.                                  */
 /*                                                                  */
 /*      Licensed Materials - Property of IBM                       */
 /*      5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,2007.      */
 /*      All Rights Reserved.                                       */
 /*      US Government Users Restricted Rights-- Use, duplication or */
 /*      disclosure restricted by GSA ADP Schedule Contract with    */
 /*      IBM Corp.                                                  */
 /*                                                                  */
 /*   DISCLAIMER OF WARRANTIES                                       */
 /*      The following "enclosed" code is sample code created by IBM */
 /*      Corporation. This sample code is not part of any standard  */
 /*      IBM product and is provided to you solely for the purpose of */
 /*      assisting you in the development of your applications.  The */
 /*      code is provided "AS IS", without warranty of any kind.    */
 /*      IBM shall not be liable for any damages arising out of your */
 /*      use of the sample code, even if IBM has been advised of the */
 /*      possibility of such damages.                               */
 /*                                                                  */
 /********************************************************************/


 /********************************************************************/
 /*                                                                  */
 /*  During initialization, IBMUEXIT is called.  It reads           */
 /*  information about the messages being screened from a text       */
 /*  file and stores the information in a hash table.  IBMUEXIT      */
 /*  also sets up the entry points for the message filter service   */
 /*  and termination service.                                        */
 /*                                                                  */
 /*  For each message generated by the compiler, the compiler        */
 /*  calls the message filter registered by IBMUEXIT.  The filter    */
 /*  looks the message up in the hash table previously created.      */
 /*                                                                  */
 /*  The termination service is called at the end of the compile     */
 /*  but does nothing.  It could be enhanced to generates reports    */
 /*  or do other cleanup work.                                       */
 /*                                                                  */
 /********************************************************************/
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 1 of 7)

```
pack: package exports(*);

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

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

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

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

      2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs  */
      2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs  */

      2 Uex_UIB_Exit_Routs,                    /* exit entries setat
                                                  initialization     */
        3 ( Uex_UIB_Termination,
            Uex_UIB_Message_Filter,            /* call for each msg  */
            *, *, *, * )
          limited entry (
            *,                                 /* to Uex_UIB         */
            *                                  /* to a request area  */
          );

  /******************************************************************/
  /*                                                                */
  /*    Request Area for Initialization exit                        */
  /*                                                                */
  /******************************************************************/

  Dcl 1 Uex_ISA native based( null() ),
        2 Uex_ISA_Length fixed bin(31);


  /******************************************************************/
  /*                                                                */
  /*    Request Area for Message_Filter exit                        */
  /*                                                                */
  /******************************************************************/

  Dcl 1 Uex_MFA native based( null() ),
        2 Uex_MFA_Length    fixed bin(31),
        2 Uex_MFA_Facility_Id  char(3),
        2 *                 char(1),
        2 Uex_MFA_Message_no    fixed bin(31),
        2 Uex_MFA_Severity     fixed bin(15),
        2 Uex_MFA_New_Severity fixed bin(15); /* set by exit proc   */

  /******************************************************************/
  /*                                                                */
  /*    Request Area for Terminate exit                             */
  /*                                                                */
  /******************************************************************/

  Dcl 1 Uex_TSA native based( null() ),
        2 Uex_TSA_Length fixed bin(31);
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 2 of 7)

```
/******************************************************************/
/*                                                                */
/*   Severity Codes                                               */
/*                                                                */
/******************************************************************/

dcl uex_Severity_Normal           fixed bin(15) value(0);
dcl uex_Severity_Warning          fixed bin(15) value(4);
dcl uex_Severity_Error            fixed bin(15) value(8);
dcl uex_Severity_Severe           fixed bin(15) value(12);
dcl uex_Severity_Unrecoverable    fixed bin(15) value(16);


/******************************************************************/
/*                                                                */
/*   Return Codes                                                 */
/*                                                                */
/******************************************************************/

dcl uex_Return_Normal             fixed bin(15) value(0);
dcl uex_Return_Warning            fixed bin(15) value(4);
dcl uex_Return_Error              fixed bin(15) value(8);
dcl uex_Return_Severe             fixed bin(15) value(12);
dcl uex_Return_Unrecoverable      fixed bin(15) value(16);


/******************************************************************/
/*                                                                */
/*   Reason Codes                                                 */
/*                                                                */
/******************************************************************/

dcl uex_Reason_Output             fixed bin(15) value(0);
dcl uex_Reason_Suppress           fixed bin(15) value(1);

dcl hashsize fixed bin(15) value(97);
dcl hashtable(0:hashsize-1) ptr init((hashsize) null());

dcl 1 message_item native based,
      2 message_Info,
        3 facid  char(3),
        3 msgno  fixed bin(31),
        3 newsev fixed bin(15),
        3 reason fixed bin(31),
      2 link pointer;
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 3 of 7)

```
ibmuexit: proc ( ue, ia  )
  options( fetchable nodescriptor );

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 ia like uex_Isa byaddr;

  dcl sysuexit     file stream input env(recsize(80));
  dcl p            pointer;
  dcl bucket       fixed bin(31);
  dcl based_Chars  char(8) based;
  dcl title_Str    char(8) var;

  ue.uex_Uib_Message_Filter = message_Filter;
  ue.uex_Uib_Termination = exitterm;


  on undefinedfile(sysuexit)
  begin;
    put edit ('** User exit unable to open exit file ')
             (A) skip;
    put skip;
    signal error;
  end;

  if ue.uex_Uib_User_Char_Len = 0 then
    do;
      open file(sysuexit);
    end;
  else
    do;
      title_Str
       = substr( ue.uex_Uib_User_Char_Str->based_Chars,
                 1, ue.uex_Uib_User_Char_Len );
      open file(sysuexit) title(title_Str);
    end;

  on error, endfile(sysuexit)
    goto done;

  allocate message_item set(p);

  /**************************************************************/
  /*                                                          */
  /*  Skip header lines and read first data line              */
  /*                                                          */
  /**************************************************************/

  get file(sysuexit) list(p->message_info) skip(3);
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 4 of 7)

```
       do loop;

          /************************************************************/
          /*                                                          */
          /*  Put message information in hash table                   */
          /*                                                          */
          /************************************************************/

          bucket = mod(p->msgno, hashsize);
          p->link = hashtable(bucket);
          hashtable(bucket) = p;

          /************************************************************/
          /*                                                          */
          /*  Read next data line                                     */
          /*                                                          */
          /************************************************************/

          allocate message_item set(p);
          get file(sysuexit) skip;
          get file(sysuexit) list(p->message_info);

       end;


       /************************************************************/
       /*                                                          */
       /*  Clean up                                                */
       /*                                                          */
       /************************************************************/

       done:

       free p->message_Item;
       close file(sysuexit);

    end;

    message_Filter:
      proc ( ue, mf  )
      options( nodescriptor );

      dcl 1 ue like uex_Uib byaddr;
      dcl 1 mf like uex_Mfa byaddr;

      dcl p pointer;
      dcl bucket fixed bin(15);

      on error snap system;

      ue.uex_Uib_Reason_Code = uex_Reason_Output;
      ue.uex_Uib_Return_Code = 0;

      mf.uex_Mfa_New_Severity = mf.uex_Mfa_Severity;


       /************************************************************/
       /*                                                          */
       /*  Calculate bucket for error message                      */
       /*                                                          */
       /************************************************************/

       bucket = mod(mf.uex_Mfa_Message_No, hashsize);
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 5 of 7)

```
          /****************************************************************/
          /*                                                            */
          /*  Search bucket for error message                          */
          /*                                                            */
          /****************************************************************/

          do p = hashtable(bucket) repeat (p->link) while(p!=null())
            until (p->msgno = mf.uex_Mfa_Message_No &
                   p->facid = mf.Uex_Mfa_Facility_Id);
          end;


          if p = null() then;
          else
            do;

              /****************************************************/
              /*                                                */
              /*  Filter error based on information in has table */
              /*                                                */
              /****************************************************/

              ue.uex_Uib_Reason_Code = p->reason;
              if p->newsev < 0 then;
              else
                mf.uex_Mfa_New_Severity = p->newsev;
            end;
          end;


          exitterm:
            proc ( ue, ta  )
            options( nodescriptor );

            dcl 1 ue like uex_Uib byaddr;
            dcl 1 ta like uex_Tsa byaddr;

            ue.uex_Uib_return_Code = 0;
            ue.uex_Uib_reason_Code = 0;

          end;

       end pack;

//*********************************************************************
//* link the user exit
//*********************************************************************
//LKEDEXIT EXEC PGM=IEWL,PARM='XREF,LIST,LET,DYNAM=DLL',
//    COND=(9,LT,PLIEXIT),REGION=5000K
//SYSLIB    DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD   DD DSN=&&EXITLIB(IBMUEXIT),DISP=(NEW,PASS),UNIT=SYSDA,
//            SPACE=(TRK,(7,1,1)),DSNTYPE=LIBRARY
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(3,1)),
//            DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=X
//SYSDEFSD DD DUMMY
//SYSLIN    DD DSN=&&LOADSET,DISP=SHR
//          DD DDNAME=SYSIN
//LKED.SYSIN    DD *
  ENTRY IBMUEXIT
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 6 of 7)

```
//********************************************************************
//* compile main
//********************************************************************
//PLI EXEC PGM=IBMZPLI,PARM='F(I),EXIT',
//            REGION=256K
//STEPLIB   DD  DSN=&&EXITLIB,DISP=SHR
//          DD DSN=IBMZ.V3R7M0.SIBMZCMP,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET2,DISP=(MOD,PASS),UNIT=SYSSQ,
//            SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//            SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*process;
 MainFet: Proc Options(Main);
   /* the exit will suppress the message for the next dcl */
   dcl one_byte_integer  fixed bin(7);
 End ;
//*
//SYSUEXIT DD *
  Fac Id   Msg No   Severity   Suppress   Comment
 +--------+--------+----------+----------+-----------------------------
  'IBM'    1042      -1         1         String spans multiple lines
  'IBM'    1044      -1         1         FIXED BIN 7 mapped to 1 byte
```

Figure 16. Sample JCL to compile, link, and invoke the user exit (Part 7 of 7)

Fetching z/OS C routines

Unless the NORENT option has been specified, the ENTRY declaration in the routine that fetches an z/OS C routine must not specify OPTIONS(COBOL) or OPTIONS(ASM)—these should be specified only for COBOL or ASM routines not linked as DLLs

The z/OS C documentation provides instructions on how to compile and link an z/OS C DLL.

Fetching assembler routines

Unless the NORENT option has been specified, the ENTRY declaration in the routine that fetches an assembler routine must specify OPTIONS(ASM).

Invoking MAIN under z/OS UNIX

Under z/OS UNIX, if you compile a MAIN program with the SYSTEM(MVS) option, the program will be passed, as usual, one CHARACTER VARYING string containing the parameters specified when it was invoked.

However, under z/OS UNIX, if you compile a MAIN program with the SYSTEM(OS) option, the program will be passed 7 parameters as specified in the z/OS UNIX manuals. These 7 parameters include:

- the argument count (which includes the name of the executable as the first "argument"
- the address of an array of addresses of the lengths of the arguments
- the address of an array of addresses of the arguments as null-terminated character strings
- the count of environment variables set

- the address of an array of addresses of the lengths of the environment variables
- the address of an array of addresses of the environment variables as null-terminated character strings

The program in Figure 17 uses the SYSTEM(OS) interface to address and display the individual arguments and environment variables.

```
*process display(std) system(os);

 sayargs:
  proc(argc, pArgLen, pArgStr, envc, pEnvLen, pEnvStr, pParmSelf)
  options( main, noexecops );

   dcl argc                 fixed bin(31) nonasgn byaddr;
   dcl pArgLen              pointer nonasgn byvalue;
   dcl pArgStr              pointer nonasgn byvalue;
   dcl envc                 fixed bin(31) nonasgn byaddr;
   dcl pEnvLen              pointer nonasgn byvalue;
   dcl pEnvStr              pointer nonasgn byvalue;
   dcl pParmSelf            pointer nonasgn byvalue;

   dcl q(4095)              pointer based;
   dcl bxb                  fixed bin(31) based;
   dcl bcz                  char(31) varz based;

   display( 'argc = ' || argc );
   do jx = 1 to argc;
     display( 'pargStr(jx) =' || pArgStr->q(jx)->bcz );
   end;
   display( 'envc = ' || envc );
   do jx = 1 to envc;
     display( 'pEnvStr(jx) =' || pEnvStr->q(jx)->bcz );
   end;

 end;
```

Figure 17. Sample program to display z/OS UNIX args and environment variables

Part 2. Using I/O facilities

Chapter 6. Using data sets and files

Your PL/I programs process and transmit units of information called *records*. A collection of records is called a *data set*. Data sets are physical collections of information external to PL/I programs; they can be created, accessed, or modified by programs written in PL/I or other languages or by the utility programs of the operating system.

Your PL/I program recognizes and processes information in a data set by using a symbolic or logical representation of the data set called a *file*. This chapter describes how to associate data sets with the files known within your program. It introduces the five major types of data sets, how they are organized and accessed, and some of the file and data set characteristics you need to know how to specify.

Note: INDEXED implies VSAM and is supported only under batch.

Associating data sets with files under z/OS

A file used within a PL/I program has a *PL/I file name*. The physical data set external to the program has a name by which it is known to the operating system: a *data set name* or *dsname*. In some cases the data set has no name; it is known to the system by the device on which it exists.

The operating system needs a way to recognize which physical data set is referred to by your program, so you must write a *data definition* or *DD* statement, external to your program, that associates the PL/I file name with a dsname. For example, if you have the following file declaration in your program:

```
DCL STOCK FILE STREAM INPUT;
```

you should create a DD statement with a *data definition name* (*ddname*) that matches the name of the PL/I file. The DD statement specifies a physical data set name (dsname) and gives its characteristics:

```
//GO.STOCK  DD DSN=PARTS.INSTOCK, . . .
```

You'll find some guidance in writing DD statements in this manual, but for more detail refer to the job control language (JCL) manuals for your system.

There is more than one way to associate a data set with a PL/I file. You associate a data set with a PL/I file by ensuring that the ddname of the DD statement that defines the data set is the same as one of the following:

- The declared PL/I file name
- The character-string value of the expression specified in the TITLE option of the associated OPEN statement.

You must choose your PL/I file names so that the corresponding ddnames conform to the following restrictions:

- If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that explicitly opens the file, the ddname defaults to the file name. If the file name is longer than 8 characters, the default ddname is composed of the first 8 characters of the file name.
- The character set of the JCL does not contain the break character (_). Consequently, this character cannot appear in ddnames. Do not use break

characters among the first 8 characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters $, @, and #, however, are valid for ddnames, but the first character must be one of the letters A through Z.

Since external names are limited to 7 characters, an external file name of more than 7 characters is shortened into a concatenation of the first 4 and the last 3 characters of the file name. Such a shortened name is **not**, however, the name used as the ddname in the associated DD statement.

Consider the following statements:

1. `OPEN FILE(MASTER);`
2. `OPEN FILE(OLDMASTER);`
3. `READ FILE(DETAIL) ...;`

When statement number 1 is run, the file name MASTER is taken to be the same as the ddname of a DD statement in the current job step. When statement number 2 is run, the name OLDMASTE is taken to be the same as the ddname of a DD statement in the current job step. (The first 8 characters of a file name form the ddname. If OLDMASTER is an external name, it will be shortened by the compiler to OLDMTER for use within the program.) If statement number 3 causes implicit opening of the file DETAIL, the name DETAIL is taken to be the same as the ddname of a DD statement in the current job step.

In each of the above cases, a corresponding DD statement must appear in the job stream; otherwise, the UNDEFINEDFILE condition is raised. The three DD statements could start as follows:

1. `//MASTER   DD ...`
2. `//OLDMASTE DD ...`
3. `//DETAIL   DD ...`

If the file reference in the statement which explicitly or implicitly opens the file is not a file constant, the DD statement name **must** be the same as the value of the file reference. The following example illustrates how a DD statement should be associated with the value of a file variable:

```
DCL PRICES FILE VARIABLE,
    RPRICE FILE;
      PRICES = RPRICE;
      OPEN FILE(PRICES);
```

The DD statement should associate the data set with the file constant RPRICE, which is the value of the file variable PRICES, thus:

```
//RPRICE DD DSNAME=...
```

Use of a file variable also allows you to manipulate a number of files at various times by a single statement. For example:

```
DECLARE F FILE VARIABLE,
        A FILE,
        B FILE,
        C FILE;
           .
           .
           .
      DO F=A,B,C;
        READ FILE (F) ...;
```

```
            .
            .
            .
        END;
```

The READ statement reads the three files A, B, and C, each of which can be associated with a different data set. The files A, B, and C remain open after the READ statement is executed in each instance.

The following OPEN statement illustrates use of the TITLE option:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

For this statement to be executed successfully, you must have a DD statement in the current job step with DETAIL1 as its ddname. It could start as follows:

```
//DETAIL1 DD DSNAME=DETAILA,...
```

Thus, you associate the data set DETAILA with the file DETAIL through the ddname DETAIL1.

Associating several files with one data set

You can use the TITLE option to associate two or more PL/I files with the same external data set, provided the first file association is closed before opening a second file association against the same TITLE name. The following example, where INVNTRY is the name of a DD statement defining a data set to be associated with two files:

```
OPEN FILE (FILE1) TITLE('INVNTRY');
.....
CLOSE FILE (FILE1);
.....
OPEN FILE (FILE2) TITLE('INVNTRY');
```

If the file is not closed first before the second open is done, the UNDEFINEDFILE condition will be raised with a subcode1 value of 59, stating that an open was attempted against a file that was already open.

Associating several data sets with one file

The file name can, at different times, represent entirely different data sets. In the above example of the OPEN statement, the file DETAIL1 is associated with the data set named in the DSNAME parameter of the DD statement DETAIL1. If you closed and reopened the file, you could specify a different ddname in the TITLE option to associate the file with a different data set.

Use of the TITLE option allows you to choose dynamically, at open time, one among several data sets to be associated with a particular file name. Consider the following example:

```
DO IDENT='A','B','C';
   OPEN FILE(MASTER)
        TITLE('MASTER1'||IDENT);
    .
    .
    .
   CLOSE FILE(MASTER);
END;
```

In this example, when MASTER is opened during the first iteration of the do-group, the associated ddname is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the ddname. During the second

iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the ddname MASTER1B. Similarly, during the final iteration of the do-group, MASTER is associated with the ddname MASTER1C.

Concatenating several data sets

For input only, you can concatenate two or more sequential or regional data sets (that is, link them so that they are processed as one continuous data set) by omitting the ddname from all but the first of the DD statements that describe them. For example, the following DD statements cause the data sets LIST1, LIST2, and LIST3 to be treated as a single data set for the duration of the job step in which the statements appear:

```
//GO.LIST DD DSNAME=LIST1,DISP=OLD
//        DD DSNAME=LIST2,DISP=OLD
//        DD DSNAME=LIST3,DISP=OLD
```

When read from a PL/I program, the concatenated data sets need not be on the same volume. You cannot process concatenated data sets backward.

Accessing HFS files under z/OS

You can access HFS files from a batch program by specifying the HFS file name in the DD statement or in the TITLE option of the OPEN statement.

For example, to access the HFS file /u/USER/sample.txt via the DD HFS, you would code the DD statement as follows:

```
//HFS DD PATH='/u/USER/sample.txt',PATHOPTS=ORDONLY,DSNTYPE=HFS
```

To access the same file by using the TITLE option of the OPEN statement, you would code:

```
OPEN FILE(HFS) TITLE('//u/USER/sample.txt');
```

Note the two forward slashes in the TITLE option: the first indicates that what follows is a file name (rather than a DD name), and the second is the start of the fully qualified HFS file name (and fully qualified names have to be used when HFS files are referenced under batch since there is no "current directory" that could be used to complete a file specification).

This is the order in which PL/I decides how to treat HFS files under batch:
- if ENV(F) was specified in the file declaration, the file will be assumed to consist of fixed length records
- if ENV(V) was specified in the file declaration, the file will be assumed to consist of lf-delimited records
- if FILEDATA=BINARY was specified on the file's DD statement, the file will be assumed to consist of fixed length records
- otherwise the file will be assumed to consist of lf-delimited records

Associating data sets with files under z/OS UNIX

A file used within a PL/I program has a PL/I file name. A data set also has a name by which it is known to the operating system.

PL/I needs a way to recognize the data 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). If the filename refers to an HFS file, the filename has to be properly qualified. Otherwise, the PL/I library will assume the filename refers to an MVS dataset.

For example:
```
declare MyFile stream output;
```

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

/datapath/mydata.dat refers to an HFS file. The filename is fully-qualified.

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

./mydata.dat refers to an HFS file residing in the current directory.

```
export DD_MYFILE=mydata.dat
```

mydata.dat refers to an MVS dataset.

If you are familiar with the IBM mainframe environment, you can think of the environment variable much like you do the:
>   DD statement in z/OS
>   ALLOCATE statement in TSO

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 165.

Under z/OS UNIX, where more types of varying length HFS files are supported than under batch, PL/I decides how to treat an HFS file as follows:
- if ENV(F) was specified in the file declaration, the file will be assumed to consist of fixed length records
- if a TYPE was specified in the file's EXPORT statement, the file will be assumed to consist of records of that type
- otherwise the file will be assumed to consist of lf-delimited records

Using the TITLE option of the OPEN statement

You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file, and, optionally, to provide additional characteristics of that data set.

```
►►──TITLE──(expression)──────────────────────────────────────►◄
```

The **expression** must yield a character string with the following syntax:

```
►►──┬─alternate_ddname──────────────────────┬──────────────────────►◄
    │                  ┌──────────────────┐ │
    └─/filespec────────┴──────────────────┴─┘
                       └─,──dd_option──┘
```

alternate_ddname
>   The name of an alternate DD_DDNAME environment variable. An alternate DD_DDNAME environment variable is one not named after a file constant. For example, if you had a file named INVENTRY in your program, and you establish two DD_DDNAME environment variables—the first named INVENTRY and the second named PARTS—you could associate the file with the second one using this statement:
>
>   ```
>   open file(Inventry) title('PARTS');
>   ```

filespec
>   Any valid file specification on the system you are using. The maximum length of filespec is 1023 characters.

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 165.

Here is an example of using the OPEN statement in this manner with a z/OS DSN:

```
open file(Payroll) title('/June.Dat,append(n),recsize(52)');
```

Note the required leading forward slash in the TITLE option. This leading forward slash indicates that what follows is a file name (rather than a DD name). In this case, June.Dat refers to an MVS dataset.

If June.Dat is an HFS file, the example would look like this:

```
open file(Payroll) title('//u/USER/June.Dat,append(n),recsize(52)');
```

Note the two forward slashes in the TITLE option: the first indicates that what follows is a file name (rather than a DD name), and the second is the start of the fully-qualified HFS file name.

Relative HFS file names can also be specified in place of fully-qualified names. In the following example:

```
open file(Payroll) title('./June.Dat,append(n),recsize(52)');
```

The dataset name *June.Dat* will be prefixed with the pathname of the current z/OS UNIX directory.

With this form, PL/I obtains all DD information either from the TITLE expression or from the ENVIRONMENT attribute of a file declaration - a DD_DDNAME environment variable is not referenced.

Attempting to use files not associated with data sets

If you attempt to use a file that has not been associated with a data set, (either through the use of the TITLE option of the OPEN statement or by establishing a DD_DDNAME environment variable), the UNDEFINEDFILE condition is raised. The only exceptions are the files SYSIN and SYSPRINT; these default to stdin and stdout, respectively.

How PL/I finds data sets

PL/I establishes the path for creating new data sets or accessing existing data sets in one of the following ways:
- The current directory.
- The paths as defined by the `export DD_DDNAME` environment variable.

Specifying characteristics using DD_DDNAME environment variables

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

The syntax of the DD_DDNAME environment variable is:

```
▶▶──DD_DDNAME=filespec─┬────────────┬──────────────────────▶◀
                       └─,──option──┘
```
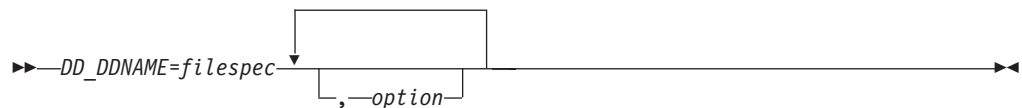
Blanks are acceptable within the syntax. In addition, the syntax of the statement is not checked at the time the command is entered. It is verified when the data set is opened. If the syntax is wrong, UNDEFINEDFILE is raised with the oncode 96.

DD_DDNAME
Specifies the name of the environment variable. The DDNAME 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 163.

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.

filespec
Specifies a file or the name of a device to be associated with the PL/I file.

option
The options you can specify as DD information.

The options that you can specify as DD information are described in the pages that follow, beginning with "APPEND" and ending with "TYPE" on page 169.

APPEND

The APPEND option specifies whether an existing data set is to be extended or recreated.

```
►►──APPEND──(──┬──Y──┬──)──────────────────────────────────────────►◄
               └──N──┘
```

Y   Specifies that new records are to be added to the end of a sequential data set, or inserted in a relative or indexed data set.

N   Specifies that, if the file exists, it is to be recreated.

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.

```
►►──ASA──(──┬──N──┬──)──────────────────────────────────────────────►◄
            └──Y──┘
```

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.

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.

BUFSIZE

The BUFSIZE option specifies the number of bytes for a buffer.

```
►►──BUFSIZE──(n)──────────────────────────────────────────────────►◄
```

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 ASCII data files as input files, and specifying the character set of output files.

```
►►──CHARSET──(──┬──ASIS───┬──)────────────────────────────────────►◄
                ├──EBCDIC─┤
                └──ASCII──┘
```

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file have (output).

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 ASCII 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 EBCDIC.

```
                  ┌─EBCDIC─┐
►►──CHARSET──(──┴─ASCII──┴──)──────────────────────────────────────►◄
```

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file to have (output).

DELAY
The DELAY option specifies the number of milliseconds to delay before retrying an operation that fails when a file or record lock cannot be obtained by the system.

```
              ┌─0─┐
►►──DELAY──(──┴─n─┴──)──────────────────────────────────────────────►◄
```

This option is applicable only to VSAM 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.

```
                 ┌─N─┐
►►──DELIMIT──(──┴─Y─┴──)────────────────────────────────────────────►◄
```

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

LRECL
The LRECL option is the same as the RECSIZE option.

```
►►──LRECL──(n)──────────────────────────────────────────────────────►◄
```

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.

```
         ┌─N─┐
►►──LRMSKIP──(─┴─Y─┴─)──────────────────────────────────────────────────────►◄
```

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.

```
        ┌─N─┐
►►──PROMPT──(─┴─Y─┴─)──────────────────────────────────────────────────────────►◄
```

PUTPAGE

The PUTPAGE option specifies whether or not the form feed character should be followed by a carriage return character. This option applies only to printer-destined files. Printer-destined files are stream output files declared with the PRINT attribute, or record output files declared with the CTLASA environment option.

```
         ┌─NOCR─┐
►►──PUTPAGE──(─┴─CR───┴─)───────────────────────────────────────────────────────►◄
```

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

CR
> Indicates that the carriage return character is appended to the form feed character. This option should be specified if output is sent to non-IBM printers.

RECCOUNT

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

```
►►──RECCOUNT──(n)────────────────────────────────────────────────────────────────►◄
```

The RECCOUNT option is ignored if PL/I does not create, or recreate, the data set.

The default for the RECCOUNT option is 50.

Note: Under z/OS, it is recommended to omit the TITLE option with both the /filespec parameter and RECCOUNT parameter for improved functionality and performance of REGIONAL(1) data sets. In such a case, the number of records that will be loaded into the file depends on the space allocated to the first extent of the data set. See Chapter 9, "Defining and using regional data sets," on page 221 for additional information.

RECSIZE

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

```
         ┌─512─┐
►►──RECSIZE──(─┴─n───┴─)─────────────────────────────────────────────────────────►◄
```

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

SAMELINE

The SAMELINE option specifies whether the system prompt occurs on the same line as the statement that prompts for input.

```
►►──SAMELINE──(──┬──N──┬──)────────────────────────────────────────►◄
                 └──Y──┘
```
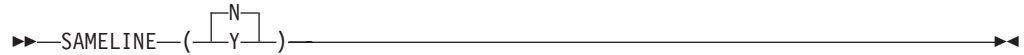
The following examples show the results of certain combinations of the PROMPT and SAMELINE options:

Example 1
> Given the statement `PUT SKIP LIST('ENTER:');`, output is as follows:

prompt(y), sameline(y)prompt(n), sameline(y)prompt(y), sameline(n) prompt(n), sameline(n)

ENTER: (cursor)ENTER: (cursor)ENTER:(cursor)ENTER:(cursor)

Example 2
> Given the statement `PUT SKIP LIST('ENTER');`, output is as follows:

prompt(y), sameline(y)prompt(n), sameline(y)prompt(y), sameline(n)   prompt(n), sameline(n)

ENTER: (cursor)ENTER (cursor)ENTER:(cursor)ENTER(cursor)

SKIP0

The SKIP0 option specifies where the line cursor moves when SKIP(0) statement is coded in the source program. SKIP0 applies to terminal files that are not linked as PM applications.

```
►►──SKIP0──(──┬──N──┬──)──────────────────────────────────────────►◄
              └──Y──┘
```
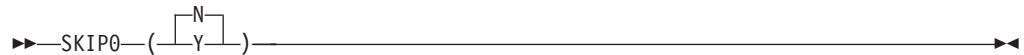
SKIP0(N)
> Specifies that the cursor is to be moved to the beginning of the next line.

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.

```
►►──TYPE──(──┬──LF──────┬──)──────────────────────────────────────►◄
             ├──CRLF────┤
             ├──TEXT────┤
             ├──FIXED───┤
             ├──CRLFEOF─┤
             └──U───────┘
```
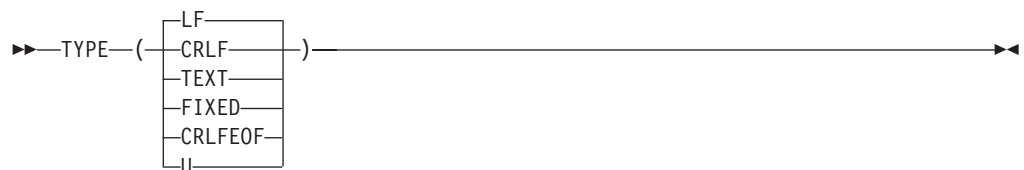
CRLF
Specifies that records are delimited by the CR - LF character combination. ('CR' and 'LF' represent the ASCII values of carriage return and line feed, '0D'x and '0A'x, respectively. For an output file, PL/I places the characters at the end of each record; for an input file, PL/I discards the characters. For both input and output, the characters are not counted in consideration for RECSIZE.

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

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

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

TEXT
Equivalent to LF.

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

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

CRLFEOF
Except for output files, this suboption specifies the same information as CRLF. When one of these files is closed for output, an end-of-file marker is appended to the last record.

U Indicates that records are unformatted. These unformatted files cannot be used by any record or stream I/O statements except OPEN and CLOSE. You can read from a TYPE(U) file only by using the FILEREAD built-in function. You can write to a TYPE(U) file only by using the FILEWRITE built-in function.

The TYPE option applies only to CONSECUTIVE files, except that it is ignored for printer-destined files with ASA(N) applied.

If your program attempts to access an existing data set with TYPE(FIXED) in effect and the length of the data set is not a multiple of the logical record length you specify, PL/I raises the UNDEFINEDFILE condition.

When using nonprint 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.

Establishing data set characteristics

A data set consists of records stored in a particular format which the operating system data management routines understand. When you declare or open a file in your program, you are describing to PL/I and to the operating system the characteristics of the records that file will contain. You can also use JCL or an

expression in the TITLE option of the OPEN statement to describe to the operating system the characteristics of the data in data sets or in the PL/I files associated with them.

You do not always need to describe your data both within the program and outside it; often one description will serve for both data sets and their associated PL/I files. There are, in fact, advantages to describing 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 will be using, you need to understand something of how the operating system moves and stores data.

Blocks and records

The items of data in a data set are arranged in blocks separated by interblock gaps (IBG). (Some manuals refer to these as interrecord gaps.)

A *block* is the unit of data transmitted to and from a data set. Each block contains one record, part of a record, or several records. You can specify the block size in the BLKSIZE parameter of the DD statement or in the BLKSIZE option of the ENVIRONMENT attribute.

A *record* is the unit of data transmitted to and from a program. You can specify the record length in the LRECL parameter of the DD statement, in the TITLE option of the OPEN statement, or in the RECSIZE option of the ENVIRONMENT attribute.

When writing a PL/I program, you need consider only the records that you are reading or writing; but when you describe the data sets that your program will create or access, you must be aware of the relationship between blocks and records.

Blocking conserves storage space in a magnetic storage volume because it reduces the number of interblock gaps, and it can increase efficiency by reducing the number of input/output operations required to process a data set. Records are blocked and deblocked by the data management routines.

Information interchange codes

The normal code in which data is recorded is the Extended Binary Coded Decimal Interchange Code (EBCDIC).

Each character in the ASCII code is represented by a 7-bit pattern and there are 128 such patterns. The ASCII set includes a substitute character (the SUB control character) that is used to represent EBCDIC characters having no valid ASCII code. The ASCII substitute character is translated to the EBCDIC SUB character, which has the bit pattern 00111111.

Record formats

The records in a data set have one of the following formats:
    Fixed-length
    Variable-length
    Undefined-length.

Records can be blocked if required. The operating system will deblock fixed-length and variable-length records, but you must provide code in your program to deblock undefined-length records.

You specify the record format in the RECFM parameter of the DD statement, in the TITLE option of the OPEN statement, or as an option of the ENVIRONMENT attribute.

Fixed-length records

You can specify the following formats for fixed-length records:

F   Fixed-length, unblocked
FB   Fixed-length, blocked
FS   Fixed-length, unblocked, standard
FBS   Fixed-length, blocked, standard.

In a data set with fixed-length records, as shown in Figure 18, all records have the same length. If the records are blocked, each block usually contains an equal number of fixed-length records (although a block can be truncated). If the records are unblocked, each record constitutes a block.

`Unblocked records (F format):`

`Blocked records (FB format):`

Figure 18. Fixed-length records

Because it bases blocking and deblocking on a constant record length, the operating system processes fixed-length records faster than variable-length records.

Variable-length records

You can specify the following formats for variable-length records:

V   Variable-length, unblocked
VB   Variable-length, blocked
VS   Variable-length, unblocked, spanned
VBS   Variable-length, blocked, spanned

V-format allows both variable-length records and variable-length blocks. A 4-byte prefix of each record and the first 4 bytes of each block contain control information for use by the operating system (including the length in bytes of the record or block). Because of these control fields, variable-length records cannot be read backward.

V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record. The first 4 bytes of the block contain block control information, and the next 4 contain record control information.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate. The first 4 bytes of the block contain block control information, and a 4-byte prefix of each record contains record control information.

Spanned Records: A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If this occurs, the record is divided into segments and accommodated in two or more consecutive blocks by specifying the record format as either VS or VBS. Segmentation and assembly are handled by the operating system. The use of spanned records allows you to select a block size, independently of record length, that will combine optimum use of auxiliary storage with maximum efficiency of transmission.

VS-format is similar to V-format. Each block contains only one record or segment of a record. The first 4 bytes of the block contain block control information, and the next 4 contain record or segment control information (including an indication of whether the record is complete or is a first, intermediate, or last segment).

VBS-format differs from VS-format in that each block contains as many complete records or segments as it can accommodate; each block is, therefore, approximately the same size (although there can be a variation of up to 4 bytes, since each segment must contain at least 1 byte of data).

Undefined-length records

U-format allows the processing of records that do not conform to F- and V-formats. The operating system and the compiler treat each block as a record; your program must perform any required blocking or deblocking.

Data set organization

The data management routines of the operating system can handle a number of types of data sets, which differ in the way data is stored within them and in the allowed means of access to the data. The three main types of non-VSAM data sets and the corresponding keywords describing their PL/I organization[1] are as follows:

Type of data set	PL/I organization
Sequential	CONSECUTIVE or ORGANIZATION(consecutive)
Indexed	INDEXED or ORGANIZATION(indexed)
Direct	REGIONAL or ORGANIZATION(relative)

A fourth type, *partitioned*, has no corresponding PL/I organization.

PL/I also provides support for three types of VSAM data organization: *ESDS*, *KSDS*, and *RRDS*. For more information about VSAM data sets, see Chapter 10, "Defining and using VSAM data sets," on page 233.

In a *sequential* (or CONSECUTIVE) data set, records are placed in physical sequence. Given one record, the location of the next record is determined by its physical position in the data set. Sequential organization can be selected for direct access devices.

An *indexed sequential* (or INDEXED) data set must reside on a direct access volume. An index or set of indexes maintained by the operating system gives the location of certain principal records. This allows direct retrieval, replacement, addition, and deletion of records, as well as sequential processing.

---

1. Do not confuse the terms "sequential" and "direct" with the PL/I file attributes SEQUENTIAL and DIRECT. The attributes refer to how the file is to be processed, and not to the way the corresponding data set is organized.

A *direct* (or REGIONAL) data set must reside on a direct access volume. The data set is divided into regions, each of which contains one or more records. A key that specifies the region number allows direct access to any record; sequential processing is also possible.

In a *partitioned* data set, independent groups of sequentially organized data, each called a member, reside in a direct access data set. The data set includes a directory that lists the location of each member. Partitioned data sets are often called *libraries*. The compiler includes no special facilities for creating and accessing partitioned data sets. Each member can be processed as a CONSECUTIVE data set by a PL/I program. The use of partitioned data sets as libraries is described under Chapter 7, "Using libraries," on page 189.

Labels

The operating system uses internal labels to identify direct access volumes and to store data set attributes (for example, record length and block size). The attribute information must originally come from a DD statement or from your program.

IBM standard labels have two parts: the initial volume label and header labels. The initial volume label identifies a volume and its owner; the header labels precede and follow each data set on the volume. Header labels contain system information, device-dependent information (for example, recording technique), and dataset characteristics.

direct access volumes have IBM standard labels. Each volume is identified by a volume label, which is stored on the volume. This label contains a volume serial number and the address of a volume table of contents (VTOC). The table of contents, in turn, contains a label, termed a *data set control block* (DSCB), for each data set stored on the volume.

Data Definition (DD) statement

A data definition (DD) statement is a job control statement that defines a data set to the operating system, and is a request to the operating system for the allocation of input/output resources. If the data sets are not dynamically allocated, each job step must include a DD statement for each data set that is processed by the step.

Your *z/OS JCL User's Guide* describes the syntax of job control statements. The operand field of the DD statement can contain keyword parameters that describe the location of the data set (for example, volume serial number and identification of the unit on which the volume will be mounted) and the attributes of the data itself (for example, record format).

The DD statement enables you to write PL/I source programs that are independent of the data sets and input/output devices they will use. You can modify the parameters of a data set or process different data sets without recompiling your program.

The following paragraphs describe the relationship of some operands of the DD statement to your PL/I program.

The LEAVE and REREAD options of the ENVIRONMENT attribute allow you to use the DISP parameter to control the action taken when the end of a magnetic-tape volume is reached or when a magnetic-tape data set is closed. The LEAVE and REREAD options are described under barry "LEAVE|REREAD" in Chapter 8.

Write validity checking, which was standard in PL/I Version 1, is no longer performed. Write validity checking can be requested through the OPTCD subparameter of the DCB parameter of the JCL DD statement. See *OS/VS2 Job Control Language* manual.

Use of the conditional subparameters

If you use the conditional subparameters of the DISP parameter for data sets processed by PL/I programs, the step abend facility must be used. The step abend facility is obtained as follows:

1. The ERROR condition should be raised or signaled whenever the program is to terminate execution after a failure that requires the application of the conditional subparameters.

2. The PL/I user exit must be changed to request an ABEND.

Data set characteristics

The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a data set, and the way it will be processed, at run time. Whereas the other parameters of the DD statement deal chiefly with the identity, location, and disposal of the data set, the DCB parameter specifies information required for the processing of the records themselves. The subparameters of the DCB parameter are described in your *z/OS JCL User's Guide.*

The DCB parameter contains subparameters that describe:

- The organization of the data set and how it will be accessed (CYLOFL, DSORG, LIMCT, NTM, and OPTCD subparameters)
- Device-dependent information such as the line spacing for a printer (CODE, FUNC, MODE, OPTCD=J, PRTSP, and STACK subparameters)
- The record format (BLKSIZE, KEYLEN, LRECL, and RECFM subparameters)
- The ASA control characters (if any) that will be inserted in the first byte of each record (RECFM subparameter).

You can specify BLKSIZE, LRECL, KEYLEN, and RECFM (or their equivalents) in the ENVIRONMENT attribute of a file declaration in your PL/I program instead of in the DCB parameter.

You cannot use the DCB parameter to override information already established for the data set in your PL/I program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied are ignored.

For a new dataset, the attributes of the file defined in the program will be used if there is a conflict with the DD statement.

You may see message IEC225I with RC=4 issued when closing PDS files. This message can be safely ignored.

An example of the DCB parameter is:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

which specifies that fixed-length records, 40 bytes in length, are to be grouped together in a block 400 bytes long.

Using the TITLE option of the OPEN statement

You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file and, optionally, to provide additional characteristics of the data set.

```
►►──TITLE──(──expression──)───────────────────────────────────────────►◄
```

The *expression* must yield a character string with the following syntax:

```
►►──┬─alternate_ddname──────────────────────────────┬──────────────────►◄
    │  ┌───────────────────────────────────────────┐ │
    │  │                                           │ │
    └──▼─/filespec─┬──────┬──┬───────────┬─────────┘
                   └──,───┘  └─dd_option─┘
```

alternate_ddname
> The name of an alternate DD_DDNAME environment variable. An alternate DD_DDNAME environment variable is one not named after a file constant. For example, if you had a file named INVENTRY in your program, and you establish two DD_DDNAME environment variables—the first named INVENTRY and the second named PARTS—you could associate the file with the second one using this statement:
> ```
> open file(Inventry) title('PARTS');
> ```

filespec
> Any valid z/OS UNIX or z/OS DSN file specification.

dd_option
> One or more options allowed in a DD_DDNAME environment variable.

For more information about options of the DD_DDNAME variable, see "Specifying characteristics using DD_DDNAME environment variables" on page 165.

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.

Associating PL/I files with data sets

Opening a file
The execution of a PL/I OPEN statement associates a file with a data set. This requires merging of the information describing the file and the data set. If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Subroutines of the PL/I library create a skeleton data control block for the data set. They use the file attributes from the DECLARE and OPEN statements and any attributes implied by the declared attributes, to complete the data control block as far as possible. (See Figure 19 on page 177.) They then issue an OPEN macro instruction, which calls the data management routines to check that the correct volume is mounted and to complete the data control block.

The data management routines examine the data control block to see what information is still needed and then look for this information, first in the DD statement, and finally, if the data set exists and has standard labels, in the data set labels. For new data sets, the data management routines begin to create the labels (if they are required) and to fill them with information from the data control block.

For INPUT data sets, the PL/I program can override the DCB attributes as long as there is no conflict in attributes. For OUTPUT data sets, the PL/I program cannot override the DCB attributes. However, if any DCB attributes are missing from the data set when it is opened, they will be obtained from the PL/I program, if provided.

When the DCB fields are filled in from these sources, control returns to the PL/I library subroutines. If any fields still are not filled in, the PL/I OPEN subroutine provides default information for some of them. For example, if LRECL is not specified, it is provided from the value given for BLKSIZE.
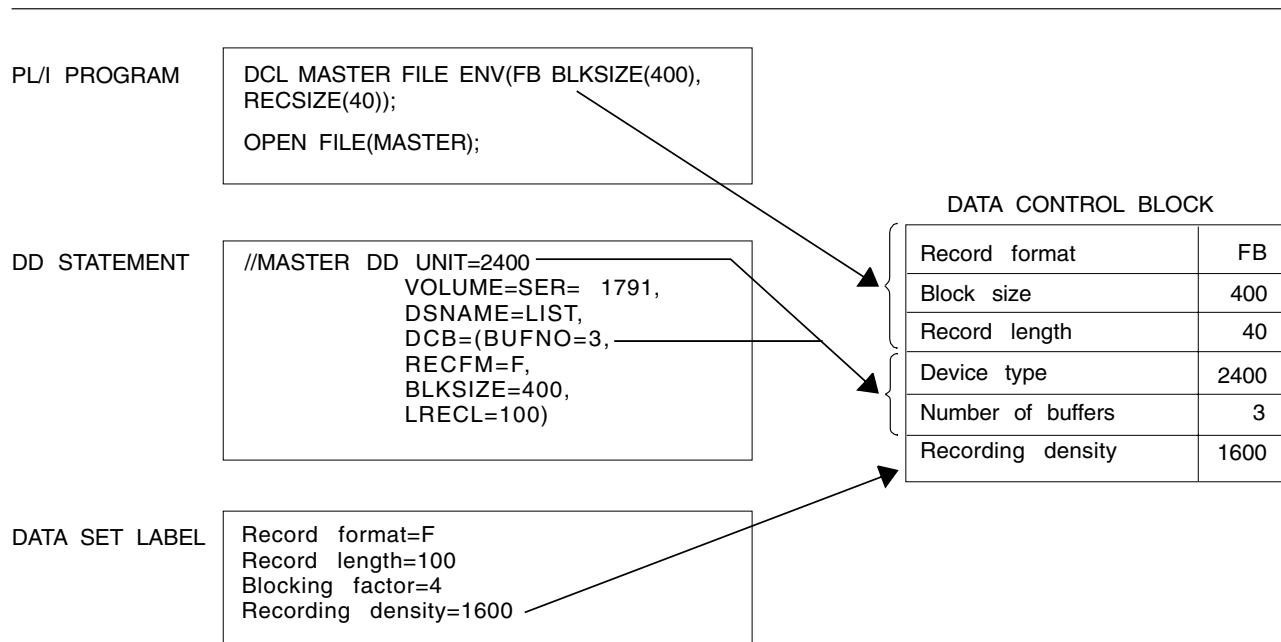
Note: Information from the PL/I program overrides that from the DD statement and the data set label.
      Information from the DD statement overrides that from the data set label.

Figure 19. How the operating system completes the DCB

Closing a file: The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated. The PL/I library subroutines first issue a CLOSE macro instruction and, when control returns from the data management routines, release the data control block that was created when the file was opened. The data management routines complete the writing of labels for new data sets and update the labels of existing data sets.

Specifying characteristics in the ENVIRONMENT attribute

You can use various options in the ENVIRONMENT attribute. Each type of file has different attributes and environment options, which are listed below.

The ENVIRONMENT attribute

You use the ENVIRONMENT attribute of a PL/I file declaration file to specify information about the physical organization of the data set associated with a file, and other related information. The format of this information must be a parenthesized option list.

►►──ENVIRONMENT──(──*option-list*──)────────────────────────────────────►◄

Abbreviation: ENV

You can specify the options in any order, separated by blanks or commas.

The following example illustrates the syntax of the ENVIRONMENT attribute in the context of a complete file declaration (the options specified are for VSAM and are discussed in Chapter 10, "Defining and using VSAM data sets," on page 233).

```
DCL FILENAME FILE RECORD SEQUENTIAL
    INPUT ENV(VSAM GENKEY);
```

Table 13 summarizes the ENVIRONMENT options and file attributes. Certain qualifications on their use are presented in the notes and comments for the figure. Those options that apply to more than one data set organization are described in the remainder of this chapter. In addition, in the following chapters, each option is described with each data set organization to which it applies.

Table 13. Attributes of PL/I file declarations

Data set type	Stream	Record									Legend:
		Consecutive		Sequential				Direct			C Checked for VSAM
											D Default
					Teleprocessing						I Must be specified or implied
											N Ignored for VSAM
File Type	Consecutive	Buffered	Unbuffered	Regional	Indexed	VSAM	Regional	Indexed	VSAM		O Optional
											S Must be specified
											- Invalid
File attributes[1]											Attributes implied
File	I	I	I	I	I	I	I	I	I	I	
Input[1]	D	D	D	D	D	D	D	D	D	D	File
Output	O	O	O	O	O	O	O	O	O	O	File
Environment	I	I	I	S	S	S	S	S	S	S	File
Stream	D	-	-	-	-	-	-	-	-	-	File
Print[1]	O	-	-	-	-	-	-	-	-	-	File stream output
Record	-	I	I	I	I	I	I	I	I	I	File
Update	-	O	O	O	-	O	O	O	O	O	File record
Sequential	-	D	D	D	-	D	D	-	-	D	File record
Buffered	-	D	-	-	I	D	D	-	-	S	File record
Keyed[2]	-	-	-	O	I	O	O	I	I	O	File record
Direct	-	-	-	-	-	-	S	S	S	S	File record keyed
ENVIRONMENT options											Comments

F|FB|FS|FBS|V|VB|VS|VBS||U	I	S	S	-	-	-	N	-	-	N	VS and VBS are invalid with Stream
F|FB|U	S	S	-	-	-	-	N	-	-	N	ASCII data sets only
F|V|U	-	-	-	S	-	-	N	S	-	N	Only F for REGIONAL(1)
F|FB|V|VB	-	-	-	-	-	S	N	-	S	N	
RECSIZE(n)	I	I	I	I	S	I	C	I	I	C	RECSIZE and/or BLKSIZE must be specified for consecutive
BLKSIZE(n)	I	I	I	I	-	I	N	I	I	N	indexed, and regional files
SCALARVARYING	-	O	O	O	-	O	O	O	O	O	Invalid for ASCII data sets
CONSECUTIVE	D	D	D	-	-	-	O	-	-	O	Allowed for VSAM ESDS
LEAVE|REREAD	O	O	O	-	-	-	-	-	-	-	
CTLASA|CTL360	-	O	O	-	-	-	-	-	-	-	Invalid for ASCII data sets
GRAPHIC	O	-	-	-	-	-	-	-	-	-	
INDEXED	-	-	-	-	-	S	O	-	S	O	Allowed for VSAM ESDS
KEYLOC(n)	-	-	-	-	-	O	-	-	O	-	
ORGANIZATION	D	-	-	-	-	-	-	-	-	-	
GENKEY	-	-	-	-	-	O	O	-	O	O	INPUT or UPDATE files only; KEYED is required
REGIONAL(1)	-	-	-	S	-	-	-	S	-	-	
VSAM	-	-	-	-	-	-	S	-	-	S	
BKWD	-	-	-	-	-	-	O	-	-	O	
REUSE	-	-	-	-	-	-	O	-	-	O	OUTPUT file only

Notes:
1. A file with the INPUT attribute cannot have the PRINT attribute.
2. Keyed is required for INDEXED and REGIONAL output.

Data set organization options: The options that specify data set organization are:

```
►►──┬─CONSECUTIVE─┬──────────────────────────────────────►◄
    ├─INDEXED─────┤
    ├─REGIONAL──(──1──)─┤
    └─VSAM────────┘
```

Each option is described in the discussion of the data set organization to which it applies.

Other ENVIRONMENT options: You can use a constant or variable with those ENVIRONMENT options that require integer arguments, such as block sizes and record lengths. The variable must not be subscripted or qualified, and must have attributes FIXED BINARY(31,0) and STATIC.

The list of equivalents for ENVIRONMENT options and DCB parameters are:

ENVIRONMENT option	DCB subparameter
Record format	RECFM[1]
RECSIZE	LRECL
BLKSIZE	BLKSIZE
CTLASA|CTL360	RECFM
KEYLENGTH	KEYLEN

Note: [1]VS must be specified as an ENVIRONMENT option, not in the DCB.

Record formats for record-oriented data transmission
Record formats supported depend on the data set organization.

```
►►─┬─F───┬────────────────────────────────────────────────►◄
   ├─FS──┤
   ├─FB──┤
   ├─FBS─┤
   ├─V───┤
   ├─VS──┤
   ├─VB──┤
   ├─VBS─┤
   └─U───┘
```

Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
	FS	unblocked, standard
	FBS	blocked, standard
Variable-length	V	unblocked
	VB	blocked
	VS	spanned
	VBS	blocked, spanned
Undefined-length	U	(cannot be blocked)

When U-format records are read into a varying-length string, PL/I sets the length of the string to the block length of the retrieved data.

These record format options do not apply to VSAM data sets. If you specify a record format option for a file associated with a VSAM data set, the option is ignored.

You can only specify VS-format records for data sets with consecutive organization.

Record formats for stream-oriented data transmission

The record format options for stream-oriented data transmission are discussed in "Using stream-oriented data transmission" on page 195.

RECSIZE option

The RECSIZE option specifies the record length.

```
►►─RECSIZE─(─record-length─)─────────────────────────────►◄
```

For files associated with VSAM data sets, **record-length** is the sum of:
1. The length required for data. For variable-length and undefined-length records, this is the maximum length.
2. Any control bytes required. Variable-length records require 4 (for the record-length prefix); fixed-length and undefined-length records do not require any.

For VSAM data sets, the maximum and average lengths of the records are specified to the Access Method Services utility when the data set is defined. If you include the RECSIZE option in the file declaration for checking purposes, you should specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

You can specify **record-length** as an integer or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:

Fixed-length, and undefined (except ASCII data sets): 32760

V-format, and VS- and VBS-format with UPDATE files: 32756

VS- and VBS-format with INPUT and OUTPUT files: 16777215

ASCII data sets: 9999

VSAM data sets: 32761

Note: For VS- and VBS-format records longer than 32,756 bytes, you must specify the length in the RECSIZE option of ENVIRONMENT, and for the DCB subparameter of the DD statement you must specify LRECL=X. If RECSIZE exceeds the allowed maximum for INPUT or OUTPUT, either a record condition occurs or the record is truncated.

Zero value:

A search for a valid value is made *first*:
* In the DD statement for the data set associated with the file, and *second*
* In the data set label.

If neither of these provides a value, default action is taken (see "Record format, BLKSIZE, and RECSIZE defaults" on page 182).

Negative Value:

The UNDEFINEDFILE condition is raised.

BLKSIZE option

The BLKSIZE option specifies the maximum block size on the data set.

```
►►──BLKSIZE──(──block-size──)──────────────────────────────────►◄
```

block-size is the sum of:

1. The total length(s) of one of the following:
   * A single record
   * A single record and either one or two record segments
   * Several records
   * Several records and either one or two record segments
   * Two record segments
   * A single record segment.

   For variable-length records, the length of each record or record segment includes the 4 control bytes for the record or segment length.

   The above list summarizes all the possible combinations of records and record segments options: fixed- or variable-length blocked or unblocked, spanned or unspanned. When specifying a blocksize for spanned records, note that each record and each record segment requires 4 control bytes for the record length and that these quantities are in addition to the 4 control bytes required for each block.

2. Any further control bytes required.
   * Variable-length blocked records require 4 (for the block size).
   * Fixed-length and undefined-length records do not require any further control bytes.

3. Any block prefix bytes required (ASCII data sets only).

block-size can be specified as an integer, or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:
> 32760

Zero value:
> If you set BLKSIZE to 0, under z/OS the Data Facility Product sets the block size. For an elaboration of this topic, see "Record format, BLKSIZE, and RECSIZE defaults."

Negative value:
> The UNDEFINEDFILE condition is raised.

The relationship of block size to record length depends on the record format:

FB-format or FBS-format
> The block size must be a multiple of the record length.

VB-format:
> The block size must be equal to or greater than the sum of:
> 1. The maximum length of any record
> 2. Four control bytes.

VS-format or VBS-format:
> The block size can be less than, equal to, or greater than the record length.

Notes:
- Use the BLKSIZE option with unblocked (F- or V-format) records in either of the following ways:
  - Specify the BLKSIZE option, but not the RECSIZE option. Set the record length equal to the block size (minus any control or prefix bytes), and leave the record format unchanged.
  - Specify both BLKSIZE and RECSIZE and ensure that the relationship of the two values is compatible with blocking for the record format you use. Set the record format to FB or VB, whichever is appropriate.
- If for FB-format or FBS-format records the block size equals the record length, the record format is set to F.
- The BLKSIZE option does not apply to VSAM data sets, and is ignored if you specify it for one.

Record format, BLKSIZE, and RECSIZE defaults
If you do not specify either the record format, block size, or record length for a non-VSAM data set, the following default action is taken:

Record format:
> A search is made in the associated DD statement or data set label. If the search does not provide a value, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets or the foreground terminal, in which case the record format is set to U.

Block size or record length:
> If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, and if this value is incompatible with the value in the specified option, the

UNDEFINEDFILE condition is raised. If the search is unsuccessful, a value is derived from the specified option (with the addition or subtraction of any control or prefix bytes).

If neither is specified, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets, in which case BLKSIZE is set to 121 for F-format or U-format records and to 129 for V-format records. For files associated with the foreground terminal, RECSIZE is set to 120.

If you are using z/OS with the Data Facility Product system-determined block size, DFP determines the optimum block size for the device type assigned. If you specify BLKSIZE(0) in either the DD assignment or the ENVIRONMENT statement, DFP calculates BLKSIZE using the record length, record format, and device type.

GENKEY option — key classification

The GENKEY (generic key) option applies only to INDEXED and VSAM key-sequenced data sets. It enables you to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

►►──GENKEY──►◄

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

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an INDEXED data set from the first nondummy record that has a key in a particular class. You identify the class by including its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although you can retrieve the first record having a key in a particular class by using a READ with the KEY option, you cannot obtain the actual key unless the records have embedded keys, since the KEYTO option cannot be used in the same statement as the KEY option.

In the following example, a key length of more than 3 bytes is assumed:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (GENKEY);
         .
         .
         .
      READ FILE(IND) INTO(INFIELD)
                   KEY ('ABC');
         .
         .
         .
NEXT:  READ FILE (IND) INTO (INFIELD);
         .
         .
         .
      GO TO NEXT;
```

The first READ statement causes the first nondummy record in the data set whose key begins with "ABC" to be read into INFIELD; each time the second READ statement is executed, the nondummy record with the next higher key is retrieved. Repeated execution of the second READ statement 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 KEYLEN subparameter. This KEYLEN subparameter is found in the DD statement that defines the indexed data set. If you specify the GENKEY option, it causes the source key to be interpreted as a generic key, and the data set is positioned to the first nondummy record in the data set whose key begins with the source key. If you do not specify the GENKEY option, a READ statement's short source key is padded on the right with blanks to the specified key length, and the data set is positioned to the record that has this padded key (if such a record exists). For a WRITE statement, a short source key is always padded with blanks.

Use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered to be the only member of its class).

SCALARVARYING option — varying-length strings

You use the SCALARVARYING option in the input/output of varying-length strings; you can use it with records of any format.

```
►►──SCALARVARYING──────────────────────────────────────────────────────────►◄
```

When storage is allocated for a varying-length string, the compiler includes a 2-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if SCALARVARYING is specified for the file.

When you use locate mode statements (LOCATE and READ SET) to create and read a data set with element varying-length strings, you must specify SCALARVARYING to indicate that a length prefix is present, since the pointer that locates the buffer is always assumed to point to the start of the length prefix.

When you specify SCALARVARYING and element varying-length strings are transmitted, you must allow two bytes in the record length to include the length prefix.

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

You must not specify SCALARVARYING and CTLASA/CTL360 for the same file, as this causes the first data byte to be ambiguous.

KEYLENGTH option

Use the KEYLENGTH option to specify the length of the recorded key for KEYED files where n is the length. You can specify KEYLENGTH for INDEXED files.

```
►►──KEYLENGTH──(──n──)──────────────────────────────────►◄
```

If you include the KEYLENGTH option in a VSAM file declaration for checking purposes, and the key length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

ORGANIZATION option

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

```
                  ┌─CONSECUTIVE─┐
►►──ORGANIZATION──(──┼─INDEXED─────┼──)──────────────────────►◄
                  └─RELATIVE────┘
```

CONSECUTIVE
Specifies that the files is associated with a consecutive data set. A consecutive file can be either a native data set or a VSAM, ESDS, RRDS, or KSDS data set.

RELATIVE
Specifies that the file is associated with a relative data set. RELATIVE specifies that the data set contains records that do not have recorded keys. A relative file is a VSAM direct data set. Relative keys range from 1 to `nnnn`.

Data set types used by PL/I record I/O

Data sets with the RECORD attribute are processed by record-oriented data transmission in which data is transmitted to and from auxiliary storage exactly as it appears in the program variables; no data conversion takes place. A record in a data set corresponds to a variable in the program.

Table 14 shows the facilities that are available with the various types of data sets that can be used with PL/I Record I/O.

Table 14. A comparison of data set types available to PL/I record I/O

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
SEQUENCE	Key order	Entry order	Num-bered	Key order	Entry order	By region
DEVICES	DASD	DASD	DASD	DASD	DASD, card, etc.	DASD
ACCESS						
1 By key 2 Sequential 3 Backward	123	123	123	12	2 3 tape only	12
Alternate index access as above	123	123	No	No	No	No

Table 14. A comparison of data set types available to PL/I record I/O (continued)

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
How extended	With new keys	At end	In empty slots	With new keys	At end	In empty slots
DELETION	Yes, 1	No	Yes, 1	Yes, 2	No	Yes, 1
1 Space reusable						
2 Space not reusable						

The following chapters describe how to use Record I/O data sets for different types of data sets:
- Chapter 8, "Defining and using consecutive data sets," on page 195
- Chapter 9, "Defining and using regional data sets," on page 221
- Chapter 10, "Defining and using VSAM data sets," on page 233

Setting environment variables

_____ **z/OS UNIX System Services Only** _____

There are a number of environment variables that can be set and exported for use with z/OS UNIX.

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

The following example illustrates how to set environment variables:

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

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

You can use the ECHO command to determine the current setting of an environment variable. To define the value of BYPASS, you can use either of the following two examples:

```
echo $LANG
```

```
echo $LIBPATH
```

_____ **End of z/OS UNIX System Services Only** _____

PL/I standard files (SYSPRINT and SYSIN)

_____ **z/OS UNIX System Services Only** _____

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. Environment variables are discussed above in "Setting environment variables" on page 186.

───────────── **End of z/OS UNIX System Services Only** ─────────────

Redirecting standard input, output, and error devices

───────────── **z/OS UNIX System Services Only** ─────────────

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
```

If you want to combine stdout and stderr in a single file, enter the following command:

```
hello2 > hello2.out 2>&1
```

As is true with display statements, the *greater than* sign redirects the output to the file that is specified after it, in this case `hello2.out`. This means that the word 'Hello' is written in the file `hello2.out`. Note also that the output includes printer control characters since the PRINT attribute is applied to SYSPRINT by default.

READ statements can access data from `stdin`; however, the record into which the data is to be put must have an LRECL equal to 1.

───────────── **End of z/OS UNIX System Services Only** ─────────────

Chapter 7. Using libraries

Within the z/OS operating system, the terms "partitioned data set", "partitioned data set/extension", and "library" are synonymous and refer to a type of data set that can be used for the storage of other data sets (usually programs in the form of source, object or load modules). A library must be stored on direct access storage and be wholly contained in one volume. It contains independent, consecutively organized data sets, called members. Each member has a unique name, not more than 8 characters long, which is stored in a directory that is part of the library. All the members of one library must have the same data characteristics because only one data set label is maintained.

You can create members individually until there is insufficient space left for a new entry in the directory, or until there is insufficient space for the member itself. You can access members individually by specifying the member name.

Use DD statements or their conversational mode equivalent to create and access members.

You can delete members by means of the IBM utility program IEHPROGM. This deletes the member name from the directory so that the member can no longer be accessed, but you cannot use the space occupied by the member itself again unless you recreate the library or compress the unused space using, for example, the IBM utility program IEBCOPY. If you attempt to delete a member by using the DISP parameter of a DD statement, it causes the whole data set to be deleted.

Types of libraries

You can use the following types of libraries with a PL/I program:
- The system program library SYS1.LINKLIB or its equivalent. This can contain all system processing programs such as compilers and the linkage editor.
- Private program libraries. These usually contain user-written programs. It is often convenient to create a temporary private library to store the load module output from the linkage editor until it is executed by a later job step in the same job. The temporary library will be deleted at the end of the job. Private libraries are also used for automatic library call by the linkage editor and the loader.
- The system procedure library SYS1.PROCLIB or its equivalent. This contains the job control procedures that have been cataloged for your installation.

How to use a library

A PL/I program can use a library directly. If you are adding a new member to a library, its directory entry will be made by the operating system when the associated file is closed, using the member name specified as part of the data set name.

If you are accessing a member of a library, its directory entry can be found by the operating system from the member name that you specify as part of the data set name.

More than one member of the same library can be processed by the same PL/I program, but only one such file can be open as output at any one time. You access different members by giving the member name in a DD statement.

Creating a library

To create a library include in your job step a DD statement containing the information given in Table 15. The information required is similar to that for a consecutively organized data set (see "Defining files using record I/O" on page 211) except for the SPACE parameter.

Table 15. Information required when creating a library

Information Required	Parameter of DD statement
Type of device that will be used	UNIT=
Serial number of the volume that will contain the library	VOLUME=SER
Name of the library	DSNAME=
Amount of space required for the library	SPACE=
Disposition of the library	DISP=

SPACE parameter

The SPACE parameter in a DD statement that defines a library must always be of the form:

```
SPACE=(units,(quantity,increment,directory))
```

Although you can omit the third term (increment), indicating its absence by a comma, the last term, specifying the number of directory blocks to be allocated, must always be present.

The amount of auxiliary storage required for a library depends on the number and sizes of the members to be stored in it and on how often members will be added or replaced. (Space occupied by deleted members is not released.) The number of directory blocks required depends on the number of members and the number of aliases. You can specify an incremental quantity in the SPACE parameter that allows the operating system to obtain more space for the data set, if such is necessary at the time of creation or at the time a new member is added; the number of directory blocks, however, is fixed at the time of creation and cannot be increased.

For example, the DD statement:

```
//   PDS DD UNIT=SYSDA,VOL=SER=3412,
//   DSNAME=ALIB,
//   SPACE=(CYL,(5,,10)),
//   DISP=(,CATLG)
```

requests the job scheduler to allocate 5 cylinders of the DASD with a volume serial number 3412 for a new library name ALIB, and to enter this name in the system catalog. The last term of the SPACE parameter requests that part of the space allocated to the data set be reserved for ten directory blocks.

Creating and updating a library member

The members of a library must have identical characteristics. Otherwise, you might later have difficulty retrieving them. Identical characteristics are necessary because the volume table of contents (VTOC) will contain only one data set control block (DSCB) for the library and not one for each member. When using a PL/I program to create a member, the operating system creates the directory entry; you cannot place information in the user data field.

When creating a library and a member at the same time, your DD statement must include all the parameters listed under "Creating a library" on page 190 (although you can omit the DISP parameter if the data set is to be temporary). The DSNAME parameter must include the member name in parentheses. For example, DSNAME=ALIB(MEM1) names the member MEM1 in the data set ALIB. If the member is placed in the library by the linkage editor, you can use the linkage editor NAME statement or the NAME compile-time option instead of including the member name in the DSNAME parameter. You must also describe the characteristics of the member (record format, etc.) either in the DCB parameter or in your PL/I program. These characteristics will also apply to other members added to the data set.

When creating a member to be added to an existing library, you do not need the SPACE parameter. The original space allocation applies to the whole of the library and not to an individual member. Furthermore, you do not need to describe the characteristics of the member, since these are already recorded in the DSCB for the library.

To add two more members to a library in one job step, you must include a DD statement for each member, and you must close one file that refers to the library before you open another.

Examples

The use of the cataloged procedure IBMZC to compile a simple PL/I program and place the object module in a new library named EXLIB is shown in Figure 20 on page 192. The DD statement that defines the new library and names the object module overrides the DD statement SYSLIN in the cataloged procedure. (The PL/I program is a function procedure that, given two values in the form of the character string produced by the TIME built-in function, returns the difference in milliseconds.)

The use of the cataloged procedure IBMZCL to compile and link-edit a PL/I program and place the load module in the existing library HPU8.CCLM is shown in Figure 21 on page 192.

```
//OPT10#1 JOB
//TR        EXEC  IBMZC
//PLI.SYSLIN  DD  UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//      SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSIN   DD  *
   ELAPSE:  PROC(TIME1,TIME2);
     DCL (TIME1,TIME2) CHAR(9),
         H1 PIC '99' DEF TIME1,
         M1 PIC '99' DEF TIME1 POS(3),
         MS1 PIC '99999' DEF TIME1 POS(5),
         H2 PIC '99' DEF TIME2,
         M2 PIC '99' DEF TIME2 POS(3),
         MS2 PIC '99999' DEF TIME2 POS(5),
         ETIME FIXED DEC(7);
     IF H2<H1 THEN H2=H2+24;
     ETIME=((H2*60+M2)*60000+MS2)-((H1*60+M1)*60000+MS1);
     RETURN(ETIME);
   END ELAPSE;
/*
```

Figure 20. Creating new libraries for compiled object modules

```
//OPT10#2  JOB
//TRLE       EXEC  IBMZCL
//PLI.SYSIN   DD  *
   MNAME:  PROC  OPTIONS(MAIN);
     .
     .
     .
     program
     .
     .
     .

   END MNAME;
/*
//LKED.SYSLMOD  DD  DSNAME=HPU8.CCLM(DIRLIST),DISP=OLD
```

Figure 21. Placing a load module in an existing library

To use a PL/I program to add or delete one or more records within a member of a
library, you must rewrite the entire member in another part of the library. This is
rarely an economic proposition, since the space originally occupied by the member
cannot be used again. You must use two files in your PL/I program, but both can
be associated with the same DD statement. The program shown in Figure 23 on
page 193 updates the member created by the program in Figure 22 on page 193. It
copies all the records of the original member except those that contain only blanks.

```
//OPT10#3  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  NMEM:  PROC OPTIONS(MAIN);
     DCL IN FILE RECORD SEQUENTIAL INPUT,
         OUT FILE RECORD SEQUENTIAL OUTPUT,
         P POINTER,
         IOFIELD CHAR(80) BASED(P),
         EOF BIT(1) INIT('0'B);
     OPEN FILE(IN),FILE (OUT);
     ON ENDFILE(IN) EOF='1'B;
     READ FILE(IN) SET(P);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
     WRITE FILE(OUT) FROM(IOFIELD);
     READ FILE(IN) SET(P);
     END;
     CLOSE FILE(IN),FILE(OUT);
  END NMEM;
/*
//GO.OUT   DD  UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//     DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//     DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
  MEM:  PROC OPTIONS(MAIN);
        /* this is an incomplete dummy library member */
```

Figure 22. Creating a library member in a PL/I program

```
//OPT10#4  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  UPDTM: PROC OPTIONS(MAIN);
     DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
         EOF BIT(1) INIT('0'B),
         DATA CHAR(80);
     ON ENDFILE(OLD)  EOF = '1'B;
     OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');
     READ FILE(OLD) INTO(DATA);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
     IF DATA=' ' THEN ;
     ELSE WRITE FILE(NEW) FROM(DATA);
     READ FILE(OLD) INTO(DATA);
     END;
   CLOSE FILE(OLD),FILE(NEW);
  END UPDTM;
/*
//GO.OLD DD  DSNAME=HPU8.ALIB(NMEM),DISP=(OLD,KEEP)
```

Figure 23. Updating a library member

Extracting information from a library directory

The directory of a library is a series of records (entries) at the beginning of the data set. There is at least one directory entry for each member. Each entry contains a member name, the relative address of the member within the library, and a variable amount of user data.

User data is information inserted by the program that created the member. An entry that refers to a member (load module) written by the linkage editor includes user data in a standard format, described in the systems manuals.

If you use a PL/I program to create a member, the operating system creates the directory entry for you and you cannot write any user data. However, you can use assembler language macro instructions to create a member and write your own user data. The method for using macro instructions to do this is described in the data management manuals.

Chapter 8. Defining and using consecutive data sets

This chapter covers consecutive data set organization and the ENVIRONMENT options that define consecutive data sets for stream and record-oriented data transmission. It then covers how to create, access, and update consecutive data sets for each type of transmission.

In a data set with consecutive organization, records are organized solely on the basis of their successive physical positions; when the data set is created, records are written consecutively in the order in which they are presented. You can retrieve the records only in the order in which they were written. See Table 13 on page 178 for valid file attributes and ENVIRONMENT options for consecutive data sets.

Using stream-oriented data transmission

This section covers how to define data sets for use with PL/I files that have the STREAM attribute. It covers the ENVIRONMENT options you can use and how to create and access data sets. The essential parameters of the DD statements you use in creating and accessing these data sets are summarized in tables, and several examples of PL/I programs are included to illustrate the text.

Data sets with the STREAM attribute are processed by stream-oriented data transmission, which allows your PL/I program to ignore block and record boundaries and treat a data set as a continuous stream of data values in character or graphic form.

You create and access data sets for stream-oriented data transmission using the list-, data-, and edit-directed input and output statements described in the *PL/I Language Reference*.

For output, PL/I converts the data items from program variables into character form if necessary, and builds the stream of characters or graphics into records for transmission to the data set.

For input, PL/I takes records from the data set and separates them into the data items requested by your program, converting them into the appropriate form for assignment to program variables.

You can use stream-oriented data transmission to read or write graphic data. There are terminals, printers, and data-entry devices that, with the appropriate programming support, can display, print, and enter graphics. You must be sure that your data is in a format acceptable for the intended device, or for a print utility program.

Defining files using stream I/O

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

```
DCL filename FILE STREAM
             INPUT | {OUTPUT [PRINT]}
             ENVIRONMENT(options);
```

195

Default file attributes are shown in Table 13 on page 178; the FILE attribute is described in the *PL/I Language Reference.* The PRINT attribute is described further in "Using PRINT files with stream I/O" on page 202. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

Table 13 on page 178 summarizes the ENVIRONMENT options. The options applicable to stream-oriented data transmission are:

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
F|FB|FS|FBS|V|VB|VS|VBS|U
RECSIZE(record-length)
BLKSIZE(block-size)
GRAPHIC
LEAVE|REREAD
```

BLKSIZE is described in Chapter 6, "Using data sets and files," beginning on page 181. LEAVE and REREAD are described later in this chapter, beginning at "LEAVE|REREAD" on page 213. Descriptions of the rest of these options follow immediately below.

CONSECUTIVE

STREAM files must have CONSECUTIVE data set organization; however, it is not necessary to specify this in the ENVIRONMENT options since CONSECUTIVE is the default data set organization. The CONSECUTIVE option for STREAM files is the same as that described in "Data set organization" on page 173.

```
►►──CONSECUTIVE──────────────────────────────────────────────────────────────►◄
```

Record format options

Although record boundaries are ignored in stream-oriented data transmission, record format is important when creating a data set. This is not only because record format affects the amount of storage space occupied by the data set and the efficiency of the program that processes the data, but also because the data set can later be processed by record-oriented data transmission.

Having specified the record format, you need not concern yourself with records and blocks as long as you use stream-oriented data transmission. You can consider your data set a series of characters or graphics arranged in lines, and you can use the SKIP option or format item (and, for a PRINT file, the PAGE and LINE options and format items) to select a new line.

```
►►──┬──F────┬──────────────────────────────────────────────────────────────►◄
    ├──FS───┤
    ├──FB───┤
    ├──FBS──┤
    ├──V────┤
    ├──VS───┤
    ├──VB───┤
    ├──VBS──┤
    └──U────┘
```

Records can have one of the following formats, which are described in "Record formats" on page 171.

Fixed-length	F	unblocked
	FB	blocked
	FS	unblocked, standard
	FBS	blocked, standard
Variable-length	V	unblocked
	VB	blocked
	VS	
	VBS	
Undefined-length	U	(cannot be blocked)

Blocking and deblocking of records are performed automatically.

RECSIZE

RECSIZE for stream-oriented data transmission is the same as that described in "Specifying characteristics in the ENVIRONMENT attribute" on page 177. Additionally, a value specified by the LINESIZE option of the OPEN statement overrides a value specified in the RECSIZE option. LINESIZE is discussed in the *PL/I Language Reference*.

Additional record-size considerations for list- and data-directed transmission of graphics are given in the *PL/I Language Reference*.

Defaults for record format, BLKSIZE, and RECSIZE

If you do not specify the record format, BLKSIZE, or RECSIZE option in the ENVIRONMENT attribute, or in the associated DD statement or data set label, the following action is taken:

Input files:
> Defaults are applied as for record-oriented data transmission, described in "Record format, BLKSIZE, and RECSIZE defaults" on page 182.

Output files:

Record format
> Set to VB-format

Record length
> The specified or default LINESIZE value is used:

PRINT files:
`F, FB, FBS,` or `U:`line size + 1
`V` or `VB:`line size + 5

Non-PRINT files:
`F, FB, FBS,` or `U:`linesize
`V` or `VB:`linesize + 4

Block size:
`F, FB,` or `FBS:`record length
`V` or `VB:`record length + 4

GRAPHIC option

Specify the GRAPHIC option for edit-directed I/O.

►►──GRAPHIC───►◄

The ERROR condition is raised for list- and data-directed I/O if you have graphics in input or output data and do not specify the GRAPHIC option.

For edit-directed I/O, the GRAPHIC option specifies that left and right delimiters are added to DBCS variables and constants on output, and that input graphics will have left and right delimiters. If you do not specify the GRAPHIC option, left and right delimiters are not added to output data, and input graphics do not require left and right delimiters. When you do specify the GRAPHIC option, the ERROR condition is raised if left and right delimiters are missing from the input data.

For information on the graphic data type, and on the G-format item for edit-directed I/O, see the *PL/I Language Reference*.

Creating a data set with stream I/O

To create a data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. For z/OS UNIX, use one of the following to provide the additional information:
* TITLE option of the OPEN statement
* DD_DDNAME environment variable
* ENVIRONMENT attribute

The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

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

  If you choose the LINESIZE option, it overrides all other sources.
* RECSIZE option of the ENVIRONMENT attribute

  The RECSIZE option of the ENVIRONMENT attribute overrides the other RECSIZE options.
* RECSIZE option of the TITLE option of the OPEN statement

  RECSIZE specified in the TITLE option of the OPEN statement has precedence over the RECSIZE option of the DD_DDNAME environment variable.
* 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 LINESIZE is not supplied, but a RECSIZE value is, PL/I derives the line-size value from RECSIZE as follows:
* For a V-format PRINT file with the ASA(N) option applied, the value is RECSIZE - 4
* For a V-format PRINT file with the ASA(Y) option applied, the value is RECSIZE - 4 - 1
* For a F-format PRINT file with the ASA(Y) option applied, the value is RECSIZE - 1
* In all other cases, 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.
- 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 V-format PRINT file with the ASA(N) option applied, the value is line size + 4
- For a V-format PRINT file with the ASA(Y) option applied, the value is line size + 4 + 1
- For a F-format PRINT file with the ASA(Y) option applied, the value is line size + 1
- In all other cases, the line-size value is assigned to the record-length value

Examples

The use of edit-directed stream-oriented data transmission to create a data set on a direct access storage device is shown in Figure 24. The data read from the input stream by the file SYSIN includes a field VREC that contains five unnamed 7-character subfields; the field NUM defines the number of these subfields that contain information. The output file WORK transmits to the data set the whole of the field FREC and only those subfields of VREC that contain information.

```
//EX7#2  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM OUTPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 PAD CHAR(25),
               2 VREC CHAR(35),
             EOF BIT(1) INIT('0'B),
             IN CHAR(80) DEF REC;
         ON ENDFILE(SYSIN) EOF='1'B;
         OPEN FILE(WORK) LINESIZE(400);
         GET FILE(SYSIN) EDIT(IN)(A(80));
         DO WHILE (¬EOF);
         PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
         GET FILE(SYSIN) EDIT(IN)(A(80));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1))
//GO.SYSIN DD *
R.C.ANDERSON      0 202848 DOCTOR
B.F.BENNETT       2 771239 PLUMBER          VICTOR HAZEL
R.E.COLE          5 698635 COOK             ELLEN  VICTOR JOAN   ANN    OTTO
J.F.COOPER        5 418915 LAWYER           FRANK  CAROL  DONALD NORMAN BRENDA
A.J.CORNELL       3 237837 BARBER           ALBERT ERIC   JANET
E.F.FERRIS        4 158636 CARPENTER        GERALD ANNA   MARY   HAROLD
/*
```

Figure 24. Creating a data set with stream-oriented data transmission

Figure 25 shows an example of a program using list-directed output to write graphics to a stream file. It assumes that you have an output device that can print graphic data. The program reads employee records and selects persons living in a certain area. It then edits the address field, inserting one graphic blank between each address item, and prints the employee number, name, and address.

```
//EX7#3  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN     DD *
% PROCESS GRAPHIC;
  XAMPLE1:  PROC OPTIONS(MAIN);
           DCL  INFILE FILE INPUT RECORD,
                OUTFILE FILE OUTPUT STREAM ENV(GRAPHIC);
 /* GRAPHIC OPTION MEANS DELIMITERS WILL BE INSERTED ON OUTPUT FILES. */
           DCL
               1 IN,
                 3 EMPNO CHAR(6),
                 3 SHIFT1 CHAR(1),
                 3 NAME,
                   5 LAST G(7),
                   5 FIRST G(7),
                 3 SHIFT2 CHAR(1),
                 3 ADDRESS,
                   5 ZIP CHAR(6),
                   5 SHIFT3 CHAR(1),
                   5 DISTRICT G(5),
                   5 CITY G(5),
                   5 OTHER G(8),
                   5 SHIFT4 CHAR(1);
           DCL EOF BIT(1) INIT('0'B);
           DCL ADDRWK G(20);
     ON ENDFILE (INFILE) EOF = '1'B;
     READ FILE(INFILE) INTO(IN);
     DO WHILE(¬EOF);
           DO;
               IF SUBSTR(ZIP,1,3)¬='300'
                 THEN LEAVE;
               L=0;
               ADDRWK=DISTRICT;
               DO I=1 TO 5;
               IF SUBSTR(DISTRICT,I,1)= <   >
                 THEN LEAVE;                /* SUBSTR BIF PICKS 3P */
               END;                         /* THE ITH GRAPHIC CHAR */
               L=L+I+1;                     /* IN DISTRICT          */
               SUBSTR(ADDRWK,L,5)=CITY;
               DO I=1 TO 5;
               IF SUBSTR(CITY,I,1)= <   >
                 THEN LEAVE;
               END;
               L=L+I;
               SUBSTR(ADDRWK,L,8)=OTHER;
               PUT FILE(OUTFILE) SKIP          /* THIS DATA SET    */
               EDIT(EMPNO,IN.LAST,FIRST,ADDRWK) /* REQUIRES UTILITY */
                   (A(8),G(7),G(7),X(4),G(20)); /* TO PRINT GRAPHIC */
                                                /* DATA             */
               END;                        /* END OF NON-ITERATIVE DO */
        READ FILE(INFILE) INTO (IN);
        END;                               /* END OF DO WHILE(¬EOF) */
     END XAMPLE1;
/*
//GO.OUTFILE   DD  SYSOUT=A,DCB=(RECFM=VB,LRECL=121,BLKSIZE=129)
//GO.INFILE DD *
ABCDEF<
>300099<       3 3 3 3 3 3 3   >
 ABCD  <
>300011<       3 3 3 3        >
 /*
```

Figure 25. Writing graphic data to a stream file

Accessing a data set with stream I/O

A data set accessed using stream-oriented data transmission need not have been created by stream-oriented data transmission, but it must have CONSECUTIVE organization, and all the data in it must be in character or graphic form. You can

open the associated file for input, and read the records the data set contains; or you can open the file for output, and extend the data set by adding records at the end.

To access a data set, you must use one of the following to identify it:
- ENVIRONMENT attribute
- DD_DDNAME environment variable
- TITLE option of the OPEN statement

The following paragraphs describe the essential information you must include in the DD statement, and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

Essential information

When your application accesses an existing STREAM file, PL/I must obtain a record-length value for that file. The value can come from one of the following sources:
- The LINESIZE option of the OPEN statement
- The RECSIZE option of the ENVIRONMENT attribute
- The RECSIZE option of the DD_DDNAME environment variable
- The RECSIZE option of the TITLE option of the OPEN statement
- PL/I-supplied default value

If you are using an existing OUTPUT file, or if you supply a RECSIZE value, PL/I determines the record-length value as described in "Creating a data set with stream I/O" on page 198.

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

Record format

When using stream-oriented data transmission to access a data set, you do not need to know the record format of the data set (except when you must specify a block size); each GET statement transfers a discrete number of characters or graphics to your program from the data stream.

If you do give record-format information, it must be compatible with the actual structure of the data set. For example, if a data set is created with F-format records, a record size of 600 bytes, and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes; but if you specify a block size of 3500 bytes, your data will be truncated.

Example

The program in Figure 26 on page 202 reads the data set created by the program in Figure 24 on page 199 and uses the file SYSPRINT to list the data it contains. (For details on SYSPRINT, see "Using SYSIN and SYSPRINT files" on page 206.) Each set of data is read, by the GET statement, into two variables: FREC, which always contains 45 characters; and VREC, which always contains 35 characters. At each execution of the GET statement, VREC consists of the number of characters generated by the expression 7*NUM, together with sufficient blanks to bring the total number of characters to 35. The DISP parameter of the DD statement could read simply DISP=OLD; if DELETE is omitted, an existing data set will not be deleted.

```
//EX7#5  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM INPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 SERNO CHAR(7),
                 3 PROF CHAR(18),
               2 VREC CHAR(35),
             IN CHAR(80) DEF REC,
             EOF BIT(1) INIT('0'B);
         ON ENDFILE(WORK) EOF='1'B;
         OPEN FILE(WORK);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
 /*
 //GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)
```

Figure 26. Accessing a data set with stream-oriented data transmission

Using PRINT files with stream I/O

Both the operating system and the PL/I language include features that facilitate
the formatting of printed output. The operating system allows you to use the first
byte of each record for a print control character. The control characters, which are
not printed, cause the printer to skip to a new line or page. (Tables of print control
characters are given in Figure 29 on page 213 and Table 17 on page 213.)

In a PL/I program, the use of a PRINT file provides a convenient means of
controlling the layout of printed output from stream-oriented data transmission.
The compiler automatically inserts print control characters in response to the
PAGE, SKIP, and LINE options and format items.

You can apply the PRINT attribute to any STREAM OUTPUT file, even if you do
not intend to print the associated data set directly. When a PRINT file is associated
with a direct access data set, the print control characters have no effect on the
layout of the data set, but appear as part of the data in the records.

PRINT files opened with FB or VB will cause the UNDEFINEDFILE condition to
be raised. PRINT files should be opened with the ″A″ option; i.e., FBA or VBA.

The compiler reserves the first byte of each record transmitted by a PRINT file for
an American National Standard print control character, and inserts the appropriate
characters automatically.

A PRINT file uses only the following five print control characters:

Character
> **Action**
> Space 1 line before printing (blank character)

0	Space 2 lines before printing
−	Space 3 lines before printing
+	No space before printing
1	Start new page

The compiler handles the PAGE, SKIP, and LINE options or format items by padding the remainder of the current record with blanks and inserting the appropriate control character in the next record. If SKIP or LINE specifies more than a 3-line space, the compiler inserts sufficient blank records with appropriate control characters to accomplish the required spacing. In the absence of a print control option or format item, when a record is full the compiler inserts a blank character (single line space) in the first byte of the next record.

If a PRINT file is being transmitted to a terminal, the PAGE, SKIP, and LINE options will never cause more than 3 lines to be skipped, unless formatted output is specified.

Controlling printed line length

You can limit the length of the printed line produced by a PRINT file either by specifying a record length in your PL/I program (ENVIRONMENT attribute) or in a DD statement, or by giving a line size in an OPEN statement (LINESIZE option). The record length must include the extra byte for the print control character, that is, it must be 1 byte larger than the length of the printed line (5 bytes larger for V-format records). The value you specify in the LINESIZE option refers to the number of characters in the printed line; the compiler adds the print control character.

The blocking of records has no effect on the appearance of the output produced by a PRINT file, but it does result in more efficient use of auxiliary storage when the file is associated with a data set on a direct access device. If you use the LINESIZE option, ensure that your line size is compatible with your block size. For F-format records, block size must be an exact multiple of (line size+1); for V-format records, block size must be at least 9 bytes greater than line size.

Although you can vary the line size for a PRINT file during execution by closing the file and opening it again with a new line size, you must do so with caution if you are using the PRINT file to create a data set on a direct access device. You cannot change the record format that is established for the data set when the file is first opened. If the line size you specify in an OPEN statement conflicts with the record format already established, the UNDEFINEDFILE condition is raised. To prevent this, either specify V-format records with a block size at least 9 bytes greater than the maximum line size you intend to use, or ensure that the first OPEN statement specifies the maximum line size. (Output destined for the printer can be stored temporarily on a direct access device, unless you specify a printer by using UNIT=, even if you intend it to be fed directly to the printer.)

Since PRINT files have a default line size of 120 characters, you need not give any record format information for them. In the absence of other information, the compiler assumes V-format records. The complete default information is:

```
BLKSIZE=129
LRECL=125
RECFM=VBA.
```

Example: Figure 27 on page 205 illustrates the use of a PRINT file and the printing options of stream-oriented data transmission statements to format a table

and write it onto a direct access device for printing on a later occasion. The table comprises the natural sines of the angles from 0° to 359° 54' in steps of 6'.

The statements in the ENDPAGE ON-unit insert a page number at the bottom of each page, and set up the headings for the following page.

The DD statement defining the data set created by this program includes no record-format information. The compiler infers the following from the file declaration and the line size specified in the statement that opens the file TABLE:

Record format =
     V (the default for a PRINT file).

Record size =
     98 (line size + 1 byte for print control character + 4 bytes for record control field).

Block size =
     102 (record length + 4 bytes for block control field).

The program in Figure 31 on page 219 uses record-oriented data transmission to print the table created by the program in Figure 27 on page 205.

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 SINE: PROC OPTIONS(MAIN);
   DCL TABLE        FILE STREAM OUTPUT PRINT;
   DCL DEG          FIXED DEC(5,1) INIT(0);  /* INIT(0) FOR ENDPAGE  */
   DCL MIN          FIXED DEC(3,1);
   DCL PGNO         FIXED DEC(2)   INIT(0);
   DCL ONCODE       BUILTIN;

   ON ERROR
     BEGIN;
       ON ERROR SYSTEM;
       DISPLAY ('ONCODE = '|| ONCODE);
     END;

   ON ENDPAGE(TABLE)
     BEGIN;
       DCL I;
       IF PGNO ¬= 0 THEN
         PUT FILE(TABLE) EDIT ('PAGE',PGNO)
             (LINE(55),COL(80),A,F(3));
       IF DEG ¬= 360 THEN
         DO;
           PUT FILE(TABLE) PAGE EDIT ('NATURAL SINES') (A);
           IF PGNO ¬= 0 THEN
             PUT FILE(TABLE) EDIT ((I DO I = 0 TO 54 BY 6))
                                  (SKIP(3),10 F(9));
           PGNO = PGNO + 1;
         END;
       ELSE
         PUT FILE(TABLE) PAGE;
     END;

   OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
   SIGNAL ENDPAGE(TABLE);

   PUT FILE(TABLE) EDIT
     ((DEG,(SIND(DEG+MIN) DO MIN = 0 TO .9 BY .1) DO DEG = 0 TO 359))
     (SKIP(2), 5 (COL(1), F(3), 10 F(9,4) ));
   PUT FILE(TABLE) SKIP(52);
 END SINE;
```

Figure 27. Creating a print file via stream data transmission. The example in Figure 31 on page 219 will print the resultant file.

Overriding the tab control table

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions. See Figure 14 on page 143 and Figure 28 on page 206 for examples of declaring a tab table. The definitions of the fields in the table are as follows:

OFFSET OF TAB COUNT:
> Halfword binary integer that gives the offset of "Tab count," the field that indicates the number of tabs to be used.

PAGESIZE:
> Halfword binary integer that defines the default page size. This page size is used for dump output to the PLIDUMP data set as well as for stream output.

LINESIZE:
> Halfword binary integer that defines the default line size.

PAGELENGTH:
> Halfword binary integer that defines the default page length for printing at a terminal.

FILLERS:
> Three halfword binary integers; reserved for future use.

TAB COUNT:
> Halfword binary integer that defines the number of tab position entries in the table (maximum 255). If tab count = 0, any specified tab positions are ignored.

Tab1–Tab*n*:
> *n* halfword binary integers that define the tab positions within the print line. The first position is numbered 1, and the highest position is numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it is ignored. The first data field in the printed output begins at the next available tab position.

You can override the default PL/I tab settings for your program by causing the linkage editor to resolve an external reference to PLITABS. To cause the reference to be resolved, supply a table with the name PLITABS, in the format described above.

To supply this tab table, include a PL/I structure in your source program with the name PLITABS, which you must declare to be STATIC EXTERNAL in your MAIN procedure or in a program linked with your MAIN procedure. An example of the PL/I structure is shown in Figure 28. 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)) FIXED BIN(15,0);
```

Figure 28. PL/I structure PLITABS for modifying the preset tab settings

Using SYSIN and SYSPRINT files

If you code a GET statement without the FILE option in your program, the compiler inserts the file name SYSIN. If you code a PUT statement without the FILE option, the compiler inserts the name SYSPRINT.

If you do not declare SYSPRINT, the compiler gives the file the attribute PRINT in addition to the normal default attributes. The complete set of attributes will be:

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

Since SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 characters and a V-format record. You need give only a minimum of information in the corresponding DD statement; if your installation uses the usual convention that the system output device of class A is a printer, the following is sufficient:

```
 //SYSPRINT DD SYSOUT=A
```

Note: SYSIN and SYSPRINT are established in the User Exit during initialization. IBM-supplied defaults for SYSIN and SYSPRINT are directed to the terminal.

You can override the attributes given to SYSPRINT by the compiler by explicitly declaring or opening the file. For more information about the interaction between SYSPRINT and the z/OS Language Environment message file option, see the *z/OS Language Environment Programming Guide*.

The compiler does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes. The data set associated with SYSIN is usually in the input stream; if it is not in the input stream, you must supply full DD information.

For more information about SYSPRINT, see "SYSPRINT considerations" on page 145.

Controlling input from the terminal

You can enter data at the terminal for an input file in your PL/I program if you do the following:
1. Declare the input file explicitly or implicitly with the CONSECUTIVE environment option (all stream files meet this condition)
2. Allocate the input file to the terminal.

You can usually use the standard default input file SYSIN because it is a stream file and can be allocated to the terminal.

You are prompted for input to stream files by a colon (:). You will see the colon each time a GET statement is executed in the program. The GET statement causes the system to go to the next line. You can then enter the required data. If you enter a line that does not contain enough data to complete execution of the GET statement, a further prompt, which is a plus sign followed by a colon (+:), is displayed.

By adding a hyphen to the end of any line that is to continue, you can delay transmission of the data to your program until you enter two or more lines.

If you include output statements that prompt you for input in your program, you can inhibit the initial system prompt by ending your own prompt with a colon. For example, the GET statement could be preceded by a PUT statement such as:

```
PUT SKIP LIST('ENTER NEXT ITEM:');
```

To inhibit the system prompt for the next GET statement, your own prompt must meet the following conditions:

1. It must be either list-directed or edit-directed, and if list-directed, must be to a PRINT file.
2. The file transmitting the prompt must be allocated to the terminal. If you are merely copying the file at the terminal, the system prompt is not inhibited.

Format of data

The data you enter at 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; the compiler will insert a comma at the end of each line.

  For instance, in response to the statement:

  ```
  GET LIST(I,J,K);
  ```

  your terminal interaction could be as follows:

  ```
  :
  1
  +:2
  +:3
  ```

  with a carriage return following each item. It would be equivalent to:

  ```
  :
  1,2,3
  ```

  If you wish to continue an item onto another line, you must end the first line with a continuation character. Otherwise, for a GET LIST or GET DATA statement, a comma will be inserted, and for a GET EDIT statement, the item will be padded (see next paragraph).

- Automatic padding for GET EDIT: There is no need to enter blanks at the end of a line of input for a GET EDIT statement. The item you enter will be padded to the correct length.

  For instance, for the PL/I statement:

  ```
  GET EDIT(NAME)(A(15));
  ```

  you could enter the five characters:

  ```
  SMITH
  ```

  followed immediately by a carriage return. The item will be padded with 10 blanks, so that the program receives a string 15 characters long. If you 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; the first line transmitted would otherwise be padded and treated as the complete data item.

- SKIP option or format item: A SKIP in a GET statement asks the program to ignore data not yet entered. All uses of SKIP(n) where n is greater than one are taken to mean SKIP(1). SKIP(1) 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 will not

necessarily be synchronized. The order in which data is transmitted to and from the terminal is not guaranteed to be the same order in which the corresponding PL/I I/O statements are executed.

Also, record file input from the terminal is received in upper case letters because of a TCAM restriction. To avoid problems you should use stream files wherever possible.

Capital and lowercase letters

For stream files, character strings are transmitted to the program as entered in lowercase or uppercase. For record files, all characters become uppercase.

End-of-file

The characters /* in positions one and two of a line that contains no other characters are treated as an end-of-file mark, that is, they raise the ENDFILE condition.

COPY option of GET statement

The GET statement can specify the COPY option; but if the COPY file, as well as the input file, is allocated to the terminal, no copy of the data will be printed.

Controlling output to the terminal

At your terminal you can obtain data from a PL/I file that has been both:
1. Declared explicitly or implicitly with the CONSECUTIVE environment option. All stream files meet this condition.
2. Allocated to the terminal.

The standard print file SYSPRINT generally meets both these conditions.

Format of PRINT files

Data from SYSPRINT or other PRINT files is not normally formatted into pages at the terminal. Three lines are always skipped for PAGE and LINE options and format items. The ENDPAGE condition is normally never raised. SKIP(n), where n is greater than three, causes only three lines to be skipped. SKIP(0) is implemented by backspacing, and should therefore not be used with terminals that do not have a backspace feature.

You can cause a PRINT file to be formatted into pages by inserting a tab control table in your program. The table must be called PLITABS, and its contents are explained in "Overriding the tab control table" on page 205. You must initialize the element PAGELENGTH to the length of page you require—that is, the length of the sheet of paper on which each page is to be printed, expressed as the maximum number of lines that could be printed on it. You must initialize the element PAGESIZE to the actual number of lines to be printed on each page. After the number of lines in PAGESIZE has been printed on a page, ENDPAGE is raised, for which standard system action is to skip the number of lines equal to PAGELENGTH minus PAGESIZE, and then start printing the next page. For other than standard layout, you must initialize the other elements in PLITABS to the values shown in Figure 14 on page 143. You can also use PLITABS to alter the tabulating positions of list-directed and data-directed output. You can use PLITABS for SYSPRINT when you need to format page breaks in ILC applications. Set PAGESIZE to 32767 and use the PUT PAGE statement to control page breaks.

Although some types of terminals have a tabulating facility, tabulating of list-directed and data-directed output is always achieved by transmission of blank characters.

Stream and record files

You can allocate both stream and record files to the terminal. However, if you allocate more than one file to the terminal and one or more is a record file, the files' output will not necessarily be synchronized. There is no guarantee that the order in which data is transmitted between the program and the terminal will be the same as the order in which the corresponding PL/I input and output statements are executed. In addition, because of a TCAM restriction, any output to record files at the terminal is printed in uppercase (capital) letters. It is therefore advisable to use stream files wherever possible.

Capital and lowercase characters

For stream files, characters are displayed at the terminal as they are held in the program, provided the terminal can display them. For instance, with an IBM 327x terminal, capital and lowercase letters are displayed as such, without translation. For record files, all characters are translated to uppercase. A variable or constant in the program can contain lowercase letters if the program was created under the EDIT command with the ASIS operand, or if the program has read lowercase letters from the terminal.

Output from the PUT EDIT command

The format of the output from a PUT EDIT command to a terminal is line mode TPUTs with "Start of field" and "end of field" characters appearing as blanks on the screen.

Using record-oriented data transmission

PL/I supports various types of data sets with the RECORD attribute (see Table 19 on page 214). This section covers how to use consecutive data sets.

Table 16 lists the statements and options that you can use to create and access a consecutive data set using record-oriented data transmission.

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	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);	

Table 16. Statements and options allowed for creating and accessing consecutive data sets (continued)

File declaration[1]	Valid statements,[2] with Options you must specify	Other options you can specify
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference) FROM(reference);	

Notes:

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

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

Specifying record format

If you give record-format information, it must be compatible with the actual structure of the data set. For example, if you create a data set with FB-format records, with a record size of 600 bytes and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes. If you specify a block size of 3500 bytes, your data is truncated.

Defining files using record I/O

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 13 on page 178. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to consecutive data sets are:

```
F|FB|FS|FBS|V|VB|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING

CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
CTLASA|CTL360
LEAVE|REREAD
```

The options through SCALARVARYING are described in "Specifying characteristics in the ENVIRONMENT attribute" on page 177, and those after SCALARVARYING are described below.

See Table 13 on page 178 to find which options you must specify, which are optional, and which are defaults.

CONSECUTIVE

The CONSECUTIVE option defines a file with consecutive data set organization, which is described in this chapter and in "Data set organization" on page 173.

```
►►──CONSECUTIVE──────────────────────────────────────────────────────────◄◄
```

CONSECUTIVE is the default.

ORGANIZATION(CONSECUTIVE)

Specifies that the file is associated with a consecutive data set. The ORGANIZATION option is described in "ORGANIZATION option" on page 185.

The file can be either a native data set or a VSAM data set.

CTLASA|CTL360

The printer control options CTLASA and CTL360 apply only to OUTPUT files associated with consecutive data sets. They specify that the first character of a record is to be interpreted as a control character.

```
►►──┬─CTLASA─┬────────────────────────────────────────────────────────────◄◄
    └─CTL360─┘
```

The CTLASA option specifies American National Standard Vertical Carriage Positioning Characters or American National Standard Pocket Select Characters (Level 1). The CTL360 option specifies IBM machine-code control characters.

The American National Standard control characters, listed in Figure 29 on page 213, cause the specified action to occur before the associated record is printed or punched.

The machine code control characters differ according to the type of device. The IBM machine code control characters for printers are listed in Table 17 on page 213.

```
Code    Action
        Space 1 line before printing (blank code)
0       Space 2 lines before printing
–       Space 3 lines before printing
+       Suppress space before printing
1       Skip to channel 1
2       Skip to channel 2
3       Skip to channel 3
4       Skip to channel 4
5       Skip to channel 5
6       Skip to channel 6
7       Skip to channel 7
8       Skip to channel 8
9       Skip to channel 9
A       Skip to channel 10
B       Skip to channel 11
C       Skip to channel 12
V       Select stacker 1
W       Select stacker 2
```

Figure 29. American National Standard print and card punch control characters (CTLASA)

Table 17. IBM machine code print control characters (CTL360)

Print and Then Act	Action	Act immediately (no printing)
Code byte		Code byte
00000001	Print only (no space)	—
00001001	Space 1 line	00001011
00010001	Space 2 lines	00010011
00011001	Space 3 lines	00011011
10001001	Skip to channel 1	10001011
10010001	Skip to channel 2	10010011
10011001	Skip to channel 3	10011011
10100001	Skip to channel 4	10100011
10101001	Skip to channel 5	10101011
10110001	Skip to channel 6	10110011
10111001	Skip to channel 7	10111011
11000001	Skip to channel 8	11000011
11001001	Skip to channel 9	11001011
11010001	Skip to channel 10	11010011
11011001	Skip to channel 11	11011011
11100001	Skip to channel 12	11100011

LEAVE|REREAD

The magnetic tape handling options LEAVE and REREAD allow you to specify the action to be taken when the end of a magnetic tape volume is reached, or when a data set on a magnetic tape volume is closed. The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to allow reprocessing of the data set. If you do not specify either of these, the action at end-of-volume or on closing of a data set is controlled by the DISP parameter of the associated DD statement.

```
 ►►──┬─LEAVE──┬─────────────────────────────────────────────────────►◄
     └─REREAD─┘
```
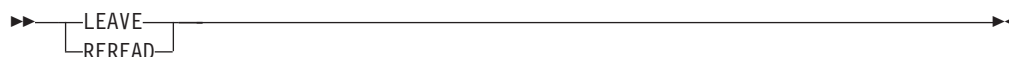
If a data set is first read or written forward and then read backward in the same program, specify the LEAVE option to prevent rewinding when the file is closed (or, with a multivolume data set, when volume switching occurs).

The effects of the LEAVE and REREAD options are summarized in Table 18.

Table 18. Effect of LEAVE and REREAD Options

ENVIRONMENT option	DISP parameter	Action
REREAD	—	Positions the current volume to reprocess the data set. Repositioning for a BACKWARDS file is at the physical end of the data set.
LEAVE	—	Positions the current volume at the logical end of the data set. Repositioning for a BACKWARDS file is at the physical beginning of the data set.
Neither REREAD nor LEAVE	PASS	Positions the volume at the end of the data set.
	DELETE	Rewinds the current volume.
	KEEP, CATLG, UNCATLG	Rewinds and unloads the current volume.

Creating a data set with record I/O

When you create a consecutive data set, you must open the associated file for SEQUENTIAL OUTPUT. You can use either the WRITE or LOCATE statement to write records. Table 16 on page 210 shows the statements and options for creating a consecutive data set.

When creating a data set, you must identify it to the operating system in a DD statement. The following paragraphs, summarized in Table 19, tell what essential information you must include in the DD statement and discuss some of the optional information you can supply.

Table 19. Creating a consecutive data set with record I/O: essential parameters of the DD statement

Storage device	When required	What you must state	Parameters
All	Always	Output device	UNIT= or SYSOUT= or VOLUME=REF=
		Block size[1]	DCB=(BLKSIZE=...
Direct access only	Always	Storage space required	SPACE=

Table 19. Creating a consecutive data set with record I/O: essential parameters of the DD statement (continued)

Storage device	When required	What you must state	Parameters
Direct access	Data set to be used by another job step but not required at end of job	Disposition	DISP=
		Disposition	DISP=
	Data set to be kept after end of job	Name of data set	DSNAME=
		Volume serial number	VOLUME=SER= or
	Data set to be on particular device		VOLUME=REF=

Notes: [1]Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

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.
- 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. See Figure 30 on page 217 for an example of a program that accesses and updates a consecutive data set. If you open the file for output, and extend the data set by adding records at the end, you must specify DISP=MOD in the DD statement. If you do not, the data set will be overwritten. If you open a file for updating, you can update only records in their existing sequence, and if you want to insert records, you must create a new data set. Table 16 on page 210 shows the statements and options for accessing and updating a consecutive data set.

When you access a consecutive data set by a SEQUENTIAL UPDATE file, you must retrieve a record with a READ statement before you can update it with a REWRITE statement; however, every record that is retrieved need not be rewritten. A REWRITE statement will always update the last record read.

Consider the following:

```
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 the operating system in a DD statement. Table 20 summarizes the DD statement parameters needed to access a consecutive data set.

Table 20. Accessing a consecutive data set with record I/O: essential parameters of the DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	
UNIT= or VOLUME=REF=	Input device	If data set not cataloged (all devices)
VOLUME=SER=	Volume serial number	If data set not cataloged (direct access)
DCB=(BLKSIZE=	Block size[1]	If data set does not have standard labels

Notes: [1]Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

The following paragraphs indicate the essential information you must include in the DD statement, and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

Essential information
If the data set is cataloged, you need to supply only the following information in the DD statement:

• The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.

• Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, opening the data set for output will result in it being overwritten.

If the data set is not cataloged, you must additionally specify the device that will read the data set and, direct access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

Example of consecutive data sets
Creating and accessing consecutive data sets are illustrated in the program in Figure 30 on page 217. The program merges the contents of two data sets, in the input stream, and writes them onto a new data set, &&TEMP; each of the original data sets contains 15-byte fixed-length records arranged in EBCDIC collating sequence. The two input files, INPUT1 and INPUT2, have the default attribute BUFFERED, and locate mode is used to read records from the associated data sets into the respective buffers. Access of based variables in the buffers should not be attempted after the file has been closed.

```
//EXAMPLE  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN   DD *
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 MERGE: PROC OPTIONS(MAIN);
   DCL (INPUT1,                             /* FIRST INPUT FILE    */
        INPUT2,                             /* SECOND INPUT FILE   */
        OUT )      FILE RECORD SEQUENTIAL;  /* RESULTING MERGED FILE*/
   DCL SYSPRINT    FILE PRINT;              /* NORMAL PRINT FILE   */

   DCL INPUT1_EOF  BIT(1) INIT('0'B);       /* EOF FLAG FOR INPUT1 */
   DCL INPUT2_EOF  BIT(1) INIT('0'B);       /* EOF FLAG FOR INPUT2 */
   DCL OUT_EOF     BIT(1) INIT('0'B);       /* EOF FLAG FOR OUT    */
   DCL TRUE        BIT(1) INIT('1'B);       /* CONSTANT TRUE       */
   DCL FALSE       BIT(1) INIT('0'B);       /* CONSTANT FALSE      */

   DCL ITEM1       CHAR(15) BASED(A);       /* ITEM FROM INPUT1    */
   DCL ITEM2       CHAR(15) BASED(B);       /* ITEM FROM INPUT2    */
   DCL INPUT_LINE  CHAR(15);                /* INPUT FOR READ INTO */
   DCL A           POINTER;                 /* POINTER VAR         */
   DCL B           POINTER;                 /* POINTER VAR         */

   ON ENDFILE(INPUT1) INPUT1_EOF = TRUE;
   ON ENDFILE(INPUT2) INPUT2_EOF = TRUE;
   ON ENDFILE(OUT)    OUT_EOF    = TRUE;

   OPEN FILE(INPUT1) INPUT,
        FILE(INPUT2) INPUT,
        FILE(OUT)    OUTPUT;

   READ FILE(INPUT1) SET(A);                /* PRIMING READ        */
   READ FILE(INPUT2) SET(B);

   DO WHILE ((INPUT1_EOF = FALSE) & (INPUT2_EOF = FALSE));
     IF ITEM1 > ITEM2 THEN
       DO;
         WRITE FILE(OUT) FROM(ITEM2);
         PUT FILE(SYSPRINT) SKIP EDIT('1>2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT2) SET(B);
       END;
     ELSE
       DO;
         WRITE FILE(OUT) FROM(ITEM1);
         PUT FILE(SYSPRINT) SKIP EDIT('1<2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT1) SET(A);
       END;
   END;
```

Figure 30. Merge Sort—creating and accessing a consecutive data set (Part 1 of 2)

```
      DO WHILE (INPUT1_EOF = FALSE);            /* INPUT2 IS EXHAUSTED  */
        WRITE FILE(OUT) FROM(ITEM1);
        PUT FILE(SYSPRINT) SKIP EDIT('1', ITEM1) (A(2),A);
        READ FILE(INPUT1) SET(A);
      END;

      DO WHILE (INPUT2_EOF = FALSE);            /* INPUT1 IS EXHAUSTED  */
        WRITE FILE(OUT) FROM(ITEM2);
        PUT FILE(SYSPRINT) SKIP EDIT('2', ITEM2) (A(2),A);
        READ FILE(INPUT2) SET(B);
      END;

      CLOSE FILE(INPUT1), FILE(INPUT2), FILE(OUT);
      PUT FILE(SYSPRINT) PAGE;
      OPEN FILE(OUT) SEQUENTIAL INPUT;

      READ FILE(OUT) INTO(INPUT_LINE);          /* DISPLAY OUT FILE     */
      DO WHILE (OUT_EOF = FALSE);
        PUT FILE(SYSPRINT) SKIP EDIT(INPUT_LINE) (A);
        READ FILE(OUT) INTO(INPUT_LINE);
      END;
      CLOSE FILE(OUT);

 END MERGE;
/*
//GO.INPUT1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.INPUT2 DD *
BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,
//         DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))
```

Figure 30. Merge Sort—creating and accessing a consecutive data set (Part 2 of 2)

The program in Figure 31 on page 219 uses record-oriented data transmission to print the table created by the program in Figure 27 on page 205.

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 PRT: PROC OPTIONS(MAIN);
    DCL TABLE        FILE RECORD INPUT SEQUENTIAL;
    DCL PRINTER      FILE RECORD OUTPUT SEQL
                        ENV(V BLKSIZE(102) CTLASA);
    DCL LINE         CHAR(94) VAR;

    DCL TABLE_EOF    BIT(1) INIT('0'B);      /* EOF FLAG FOR TABLE  */
    DCL TRUE         BIT(1) INIT('1'B);      /* CONSTANT TRUE       */
    DCL FALSE        BIT(1) INIT('0'B);      /* CONSTANT FALSE      */


    ON ENDFILE(TABLE) TABLE_EOF = TRUE;

    OPEN FILE(TABLE),
         FILE(PRINTER);

    READ FILE(TABLE) INTO(LINE);             /* PRIMING READ        */

    DO WHILE (TABLE_EOF = FALSE);
      WRITE FILE(PRINTER) FROM(LINE);
      READ FILE(TABLE) INTO(LINE);
    END;

    CLOSE FILE(TABLE),
          FILE(PRINTER);
 END PRT;
```

Figure 31. Printing record-oriented data transmission

Chapter 9. Defining and using regional data sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. How to create and access regional data sets for each type of regional organization is then 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 or more than one record, depending on the type of regional organization. The regions are numbered in succession, beginning with zero, and a record can be accessed by specifying its region number, and perhaps a key, in a data transmission statement.

Regional data sets are confined to direct access devices.

Regional organization of a data set allows you to control the physical placement of records in the data set, and to optimize the access time for a particular application. Such optimization is not available with consecutive or indexed organization, in which successive records are written either in strict physical sequence or in logical sequence depending on ascending key values; neither of these methods takes full advantage of the characteristics of direct access storage devices.

You can create a regional data set in a manner similar to a consecutive or indexed data set, presenting records in the order of ascending region numbers; alternatively, you can use direct access, in which you present records in random sequence and insert them directly into preformatted regions. 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.

The major advantage of regional organization over other types of data set organization is that it allows you to control the relative placement of records; by judicious programming, you can optimize record access in terms of device capabilities and the requirements of particular applications.

Direct access of regional data sets is quicker than that of indexed data sets, but regional data sets have the disadvantage that sequential processing can present records in random sequence; the order of sequential retrieval is not necessarily that in which the records were presented, nor need it be related to the relative key values.

Table 21 on page 222 lists the data transmission statements and options that you can use to create and access a regional data set.

221

Table 21. Statements and options allowed for creating and accessing regional data sets

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

Notes:

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

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

3. The file must not 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:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

Since BUFFERED and UNBUFFERED will be treated the same for REGIONAL(1) data sets, either option can be specified in the ENV option. For example, the FROM option is not required on a REWRITE for a SEQUENTIAL UNBUFFERED file and the LOCATE statement is allowed for OUTPUT SEQUENTIAL data sets even if UNBUFFERED is specified.

To define a direct regional data set, use a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 13 on page 178. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to regional data sets are:

```
REGIONAL({1})
F
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
```

REGIONAL option

Use the REGIONAL option to define a file with regional organization.

```
▶▶──REGIONAL──(──1──)──────────────────────────────────────────────────────◀◀
```

1   specifies REGIONAL(1)

REGIONAL(1)
   specifies that the data set contains F-format records that do not have recorded keys. Each region in the data set contains only one record; therefore, each region number corresponds to a relative record within the data set (that is, region numbers start with 0 at the beginning of the data set).

   Although REGIONAL(1) data sets have no recorded keys, you can use REGIONAL(1) DIRECT INPUT or UPDATE files to process data sets that do have recorded keys.

RECSIZE(record-length)
BLKSIZE(block-size)
   If both RECSIZE and BLKSIZE are specified, they must specify the same value.

REGIONAL(1) organization is most suited to applications where there are no duplicate region numbers, and where most of the regions will be filled (reducing wasted space in the data set).

Using keys with REGIONAL data sets

There are two kinds of keys, recorded keys and source keys. A *recorded key* is a character string that immediately precedes each record in the data set to identify that record; its length cannot exceed 255 characters. A *source key* is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. When you access a record in a regional data set, the source key gives a region number, and can also give a recorded key.

Unlike the keys for indexed data sets, recorded keys in a regional data set are never embedded within the record.

Using REGIONAL(1) data sets

In a REGIONAL(1) data set, since there are no recorded keys, the region number serves as the sole identification of a particular record. The character value of the source key should represent an unsigned decimal integer that should not exceed 16777215 (although the actual number of records allowed can be smaller, depending on a combination of record size, device capacity, and limits of your access method. For direct regional(1) files with fixed format records, the maximum number of tracks which can be addressed by relative track addressing is 65,536.) If the region number exceeds this figure, it is treated as modulo 16777216; for instance, 16777226 is treated as 10. Only the characters 0 through 9 and the blank character are valid in the source key; leading blanks are interpreted as zeros. Embedded blanks are not allowed in the number; the first embedded blank, if any, terminates the region number. If more than 8 characters appear in the source key, only the rightmost 8 are used as the region number; if there are fewer than 8 characters, blanks (interpreted as zeros) are inserted on the left.

Dummy Records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant (8)'1'B in its first byte. Although such dummy records are inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read; your PL/I program must be prepared to recognize them. You can replace dummy records with valid data. Note that if you insert (8)'1'B in the first byte, the record can be lost if you copy the file onto a data set that has dummy records that are not retrieved.

Creating a REGIONAL(1) data set

You can create a REGIONAL(1) data set either sequentially or by direct access. Table 21 on page 222 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, the opening of the file causes all tracks on the data set to be cleared, and a capacity record to be written at the beginning of each track to record the amount of space available on that track. You must present records in ascending order of region numbers; any region you omit from the sequence is filled with a dummy record. If there is an

error in the sequence, or if you present a duplicate key, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If you use a DIRECT OUTPUT file to create the data set, the whole primary extent allocated to the data set is filled with dummy records when the file is opened. You can present records in random order; if you present a duplicate, the existing record will be overwritten.

For sequential creation, the data set can have up to 15 extents, which can be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

Example

Creating a REGIONAL(1) data set is illustrated in Figure 32 on page 226. The data set is a list of telephone numbers with the names of the subscribers to whom they are allocated. The telephone numbers correspond with the region numbers in the data set, the data in each occupied region being a subscriber's name.

```
//EX9    JOB
//STEP1  EXEC IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 CRR1:   PROC OPTIONS(MAIN);
  /*  CREATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY     */

   DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
   DCL  SYSIN FILE INPUT RECORD;
   DCL  SYSIN_REC BIT(1) INIT('1'B);
   DCL 1   CARD,
       2   NAME   CHAR(20),
       2   NUMBER CHAR( 2),
       2   CARD_1 CHAR(58);
   DCL IOFIELD CHAR(20);

      ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
      OPEN FILE(NOS);
      READ FILE(SYSIN) INTO(CARD);

      DO WHILE(SYSIN_REC);
         IOFIELD = NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         READ FILE(SYSIN) INTO(CARD);
      END;

      CLOSE FILE(NOS);
   END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS    DD DSN=MYID.NOS,UNIT=SYSDA,SPACE=(20,100),
//          DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.       28
CHEESNAME,L.       11
CORY,G.            36
ELLIOTT,D.         85
FIGGINS,E.S.       43
HARVEY,C.D.W.      25
HASTINGS,G.M.      31
KENDALL,J.G.       24
LANCASTER,W.R.     64
MILES,R.           23
NEWMAN,M.W.        40
PITT,W.H.          55
ROLF,D.E.          14
SHEERS,C.D.        21
SURCLIFFE,M.       42
TAYLOR,G.C.        47
WILTON,L.W.        44
WINSTONE,E.M.      37
/*
```

Figure 32. Creating a REGIONAL(1) data set

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 21 on page 222 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 8 characters, the value returned (the 8-character region number) is padded on the left with blanks. If the target string has fewer than 8 characters, the value returned is truncated on the left.

Sequential access is in the order of ascending region numbers. All records are retrieved, whether dummy or actual, and you must ensure that your PL/I program recognizes dummy records.

Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region-number sequence, and in sequential update you can read and rewrite each record in turn.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(1) data set are identical to those for a consecutive data set. Consecutive data sets are discussed in detail in Chapter 8, "Defining and using consecutive data sets," on page 195.

Direct access

To open a DIRECT file that is used to process a REGIONAL(1) data set you can use either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

Use DIRECT UPDATE files to retrieve, add, delete, or replace records in a REGIONAL(1) data set according to the following conventions:

Retrieval
>All records, whether dummy or actual, are retrieved. Your program must recognize dummy records.

Addition
>A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key.

Deletion
>The record you specify by the source key in a DELETE statement is converted to a dummy record.

Replacement
>The record you specify by the source key in a REWRITE statement, whether dummy or actual, is replaced.

Example

Updating a REGIONAL(1) data set is illustrated in Figure 33 on page 228. 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.

```
//EX10    JOB
//STEP2   EXEC  IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 ACR1: PROC OPTIONS(MAIN);
   /*  UPDATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY       */
    DCL NOS FILE RECORD  KEYED ENV(REGIONAL(1));
    DCL  SYSIN FILE INPUT RECORD;
    DCL  (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
    DCL 1   CARD,
         2   NAME   CHAR(20),
         2   (NEWNO,OLDNO)  CHAR( 2),
         2   CARD_1 CHAR( 1),
         2   CODE   CHAR( 1),
         2   CARD_2 CHAR(54);
    DCL IOFIELD CHAR(20);
    DCL BYTE    CHAR(1) DEF IOFIELD;

    ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
    OPEN FILE (NOS) DIRECT UPDATE;
    READ FILE(SYSIN) INTO(CARD);

    DO WHILE(SYSIN_REC);
       SELECT(CODE);
          WHEN('A','C') DO;
              IF CODE = 'C' THEN
                 DELETE FILE(NOS) KEY(OLDNO);
              READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
              IF UNSPEC(BYTE) = (8)'1'B
                 THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
              ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
          END;
          WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
       OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
       END;
       READ FILE(SYSIN) INTO(CARD);
    END;

    CLOSE FILE(SYSIN),FILE(NOS);
    PUT FILE(SYSPRINT) PAGE;
    OPEN FILE(NOS) SEQUENTIAL INPUT;
    ON ENDFILE(NOS) NOS_REC = '0'B;
    READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
    DO WHILE(NOS_REC);
       IF UNSPEC(BYTE) ¬= (8)'1'B
          THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD)(A(2),X(3),A);
       PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
       READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
    END;
  CLOSE FILE(NOS);
  END ACR1;
/*
```

Figure 33. Updating a REGIONAL(1) data set (Part 1 of 2)

```
//GO.NOS    DD DSN=J44PLI.NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.          5640 C
GOODFELLOW,D.T.      89   A
MILES,R.              23 D
HARVEY,C.D.W.        29   A
BARTLETT,S.G.        13   A
CORY,G.               36 D
READ,K.M.            01   A
PITT,W.H.             55
ROLF,D.F.             14 D
ELLIOTT,D.          4285 C
HASTINGS,G.M.         31 D
BRAMLEY,O.H.        4928 C
/*
```

Figure 33. Updating a REGIONAL(1) data set (Part 2 of 2)

Essential information for creating and accessing regional data sets

To create a regional data set, you must give the operating system certain information, either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

You must supply the following information when creating a regional data set:
- Device that will write your data set (UNIT or VOLUME parameter of DD statement).
- Block size: You can specify the block size either in your PL/I program (in the BLKSIZE option of the ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size. If you do specify a record length, it must be equal to the block size.

If you want to keep a data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but do not need it after the end of your job.

If you want your data set stored on a particular direct access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system allocates one, informs the operator, and prints the number on your program listing. All the essential parameters required in a DD statement for the creation of a regional data set are summarized in Table 22 on page 230; and Table 23 on page 230 lists the DCB subparameters needed. See your *z/OS JCL User's Guide* for a description of the DCB subparameters.

You cannot place a regional data set on a system output (SYSOUT) device.

In the DCB parameter, if you specify the DSORG parameter, you must specify the data set organization as direct by coding DSORG=DA. You cannot specify the DUMMY or DSN=NULLFILE parameters in a DD statement for a regional data set.

Using DSORG=DA may cause message IEC225I to be issued. This message can be safely ignored.

Table 22. Creating a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
UNIT= or VOLUME=REF=	Output device[1]	Always
SPACE=	Storage space required[2]	
DCB=	Data control block information: see Table 23	
DISP=	Disposition	Data set to be used in another job step but not required in another job
DISP=	Disposition	Data set to be kept after end of job
DSNAME=	Name of data set	
VOLUME=SER= or VOLUME=REF=	Volume serial number	Data set to be on particular volume

[1]Regional data sets are confined to direct access devices.

[2]For sequential access, the data set can have up to 15 extents, which can be on more than one volume. For creation with DIRECT access, the data set can have only one extent.

To access a regional data set, you must identify it to the operating system in a DD statement. The following paragraphs indicate the minimum information you must include in the DD statement; this information is summarized in Table 24 on page 231.

If the data set is cataloged, you need to supply only the following information in your DD statement:
- The name of the data set (DSNAME parameter). The operating system locates the information that describes the data set in the system catalog and, if necessary, requests the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

When opening a multiple-volume regional data set for sequential update, the ENDFILE condition is raised at the end of the first volume.

Table 23. DCB subparameters for a regional data set

Subparameters	To specify	When required
RECFM=F	Record format[1]	These are always required
BLKSIZE=	Block size[1]	
DSORG=DA	Data set organization	

[1]Or you can specify the block size in the ENVIRONMENT attribute.

Table 24. Accessing a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	
UNIT= or VOLUME=REF=	Input device	If data set not cataloged
VOLUME=SER=	Volume serial number	

Chapter 10. Defining and using VSAM data sets

This chapter covers VSAM (the Virtual Storage Access Method) organization for record-oriented data transmission, VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and the statements you use to load and access the three types of VSAM data sets that PL/I supports—entry-sequenced, key-sequenced, and relative record. The chapter is concluded by a series of examples showing the PL/I statements, Access Method Services commands, and JCL statements necessary to create and access VSAM data sets.

Enterprise PL/I provides no support for ISAM datasets.

For additional information about the facilities of VSAM, the structure of VSAM data sets and indexes, the way in which they are defined by Access Method Services, and the required JCL statements, see the VSAM publications for your system.

Using VSAM data sets

How to run a program with VSAM data sets

Before you execute a program that accesses a VSAM data set, you need to know:
- The name of the VSAM data set
- The name of the PL/I file
- Whether you intend to share the data set with other users

Then you can write the required DD statement to access the data set:

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

For example, if your file is named PL1FILE, your data set named VSAMDS, and you want exclusive control of the data set, enter:

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

To share your data set, use DISP=SHR.

Enterprise PL/I has no support for ISAM data sets.

To optimize VSAM's performance by controlling the number of VSAM buffers used for your data set, see the VSAM publications.

Pairing an Alternate Index Path with a File

When using an alternate index, you simply specify the name of the *path* in the DSNAME parameter of the DD statement associating the base data set/alternate index pair with your PL/I file. Before using an alternate index, you should be aware of the restrictions on processing; these are summarized in Table 26 on page 238.

Given a PL/I file called PL1FILE and the alternate index path called PERSALPH, the DD statement required would be:

```
//PL1FILE DD DSNAME=PERSALPH,DISP=OLD
```

VSAM organization

PL/I provides support for three types of VSAM data sets:
- Key-sequenced data sets (KSDS)
- Entry-sequenced data sets (ESDS)
- Relative record data sets (RRDS).

These correspond roughly to PL/I indexed, consecutive, and regional data set organizations, respectively. They are all ordered, and they can all have keys associated with their records. Both sequential and keyed access are possible with all three types.

Although only key-sequenced data sets have keys as part of their logical records, keyed access is also possible for entry-sequenced data sets (using relative-byte addresses) and relative record data sets (using relative record numbers).

All VSAM data sets are held on direct access storage devices, and a virtual storage operating system is required to use them.

The physical organization of VSAM data sets differs from those used by other access methods. VSAM does not use the concept of blocking, and, except for relative record data sets, records need not be of a fixed length. In data sets with VSAM organization, the data items are arranged in *control intervals*, which are in turn arranged in *control areas.* For processing purposes, the data items within a control interval are arranged in logical records. A control interval can contain one or more logical records, and a logical record can span two or more control intervals. Concern about blocking factors and record length is largely removed by VSAM, although records cannot exceed the maximum specified size. VSAM allows access to the control intervals, but this type of access is not supported by PL/I.

VSAM data sets can have two types of indexes—prime and alternate. A *prime index* is the index to a KSDS that is established when you define a data set; it always exists and can be the only index for a KSDS. You can have one or more *alternate indexes* on a KSDS or an ESDS. Defining an *alternate index* for an ESDS enables you to treat the ESDS, in general, as a KSDS. An alternate index on a KSDS enables a field in the logical record different from that in the prime index to be used as the key field. Alternate indexes can be either *nonunique*, in which duplicate keys are allowed, or *unique*, in which they are not. The prime index can never have duplicate keys.

Any change in a data set that has alternate indexes must be reflected in all the indexes if they are to remain useful. This activity is known as *index upgrade*, and is done by VSAM for any index in the *index upgrade set* of the data set. (For a KSDS, the prime index is always a member of the index upgrade set.) However, you must avoid making changes in the data set that would cause duplicate keys in the prime index or in a unique alternate index.

Before using a VSAM data set for the first time, you need to define it to the system with the DEFINE command of Access Method Services, which you can use to completely define the type, structure, and required space of the data set. This command also defines the data set's indexes (together with their key lengths and locations) and the index upgrade set if the data set is a KSDS or has one or more alternate indexes. A VSAM data set is thus "created" by Access Method Services.

The operation of writing the initial data into a newly created VSAM data set is referred to as *loading* in this publication.

Use the three different types of data sets according to the following purposes:

- Use *entry-sequenced data sets* for data that you primarily access in the order in which it was created (or the reverse order).
- Use *key-sequenced data sets* when you normally access records through keys within the records (for example, a stock-control file where the part number is used to access a record).
- Use *relative record data sets* for data in which each item has a particular number, and you normally access the relevant record by that number (for example, a telephone system with a record associated with each number).

You can access records in all types of VSAM data sets either directly by means of a key, or sequentially (backward or forward). You can also use a combination of the two ways: Select a starting point with a key and then read forward or backward from that point.

You can create *alternate indexes* for key-sequenced and entry-sequenced data sets. You can then access your data in many sequences or by one of many keys. For example, you could take a data set held or indexed in order of employee number and index it by name in an *alternate index*. Then you could access it in alphabetic order, in reverse alphabetic order, or directly using the name as a key. You could also access it in the same kind of combinations by employee number.

Table 25 shows how the same data could be held in the three different types of VSAM data sets and illustrates their respective advantages and disadvantages.

Table 25. Types and advantages of VSAM data sets

Data set type	Method of loading	Method of reading	Method of updating	Pros and cons
Key-Sequenced	Sequentially in order or prime index which must be unique	KEYED by specifying key of record in prime index SEQUENTIAL backward or forward in order of any index Positioning by key followed by sequential reading either backward or forward	KEYED specifying a unique key in any index SEQUENTIAL following positioning by unique key Record deletion allowed Record insertion allowed	*Advantages*: Complete access and updating *Disadvantages*: Records must be in order of prime index before loading *Uses*: For uses where access will be related to key
Entry-Sequenced	Sequentially (forward only) The RBA of each record can be obtained and used as a key	SEQUENTIAL backward or forward KEYED using RBA Positioning by key followed by sequential either backward or forward	New records at end only Existing records cannot have length changed Record deletion not allowed	*Advantages*: Simple fast creation No requirement for a unique index *Disadvantages*: Limited updating facilities *Uses*: For uses where data will primarily be accessed sequentially

Table 25. Types and advantages of VSAM data sets (continued)

Data set type	Method of loading	Method of reading	Method of updating	Pros and cons
Relative Record	Sequentially starting from slot 1 KEYED specifying number of slot Positioning by key followed by sequential writes	KEYED specifying numbers as key Sequential forward or backward omitting empty records	Sequentially starting at a specified slot and continuing with next slot Keyed specifying numbers as key Record deletion allowed Record insertion into empty slots allowed	*Advantages*: Speedy access to record by number *Disadvantages*: Structure tied to numbering sequences Fixed length records *Uses*: For use where records will be accessed by number

Keys for VSAM data sets

All VSAM data sets can have keys associated with their records. For key-sequenced data sets, and for entry-sequenced data sets accessed via an *alternate index*, the key is a defined field within the logical record. For entry-sequenced data sets, the key is the *relative byte address* (RBA) of the record. For relative-record data sets, the key is a *relative record number*.

Keys for indexed VSAM data sets

Keys for key-sequenced data sets and for entry-sequenced data sets accessed via an *alternate index* are part of the logical records recorded on the data set. You define the length and location of the keys when you create the data set.

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 Chapter 12 of the *PL/I Language Reference.*

Relative byte addresses (RBA)

Relative byte addresses allow you to use keyed access on an ESDS associated with a KEYED SEQUENTIAL file. The RBAs, or keys, are character strings of length 4, and their values are defined by VSAM. You cannot construct or manipulate RBAs in PL/I; you can, however, compare their values in order to determine the relative positions of records within the data set. RBAs are not normally printable.

You can obtain the RBA for a record by using the KEYTO option, either on a WRITE statement when you are loading or extending the data set, or on a READ statement when the data set is being read. You can subsequently use an RBA obtained in either of these ways in the KEY option of a READ or REWRITE statement.

Do not use an RBA in the KEYFROM option of a WRITE statement.

VSAM allows use of the relative byte address as a key to a KSDS, but this use is not supported by PL/I.

Relative record numbers

Records in an RRDS are identified by a relative record number that starts at 1 and 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 8. The character value of a source key you use in the KEY or KEYFROM option must represent an unsigned integer. If the source key is not 8 characters long, it is truncated or padded with blanks (interpreted as zeros) on the left. The value returned by the KEYTO option is a character string of length 8, with leading zeros suppressed.

Choosing a data set type

When planning your program, the first decision to be made is which type of data set to use. There are three types of VSAM data sets and five types of non-VSAM data sets available to you. VSAM data sets can provide all the function of the other types of data sets, plus additional function available only in VSAM. VSAM can usually match other data set types in performance, and often improve upon it. However, VSAM is more subject to performance degradation through misuse of function.

The comparison of all eight types of data sets given in Table 14 on page 185 is helpful; however, many factors in the choice of data set type for a large installation are beyond the scope of this book.

When choosing between the VSAM data set types, you should base your choice on the most common sequence in which you will require your data. The following is a suggested procedure that you can use to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and how it will be accessed.
   a. Primarily sequentially — favors ESDS.
   b. Primarily by key — favors KSDS.
   c. Primarily by number — favors RRDS.
2. Determine how you will load the data set. Note that you must load a KSDS in key sequence; thus an ESDS with an *alternate index* path can be a more practical alternative for some applications.
3. Determine whether you require access through an *alternate index* path. These are only supported on KSDS and ESDS. If you require an *alternate index* path, determine whether the *alternate index* will have unique or nonunique keys. Use of nonunique keys can limit key processing. However, it might also be impractical to assume that you will use unique keys for all future records; if you attempt to insert a record with a nonunique key in an index that you have created for unique keys, it will cause an error.
4. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported. Figure 34 on page 238 might be helpful.

Do not try to access a dummy VSAM data set, because you will receive an error message indicating that you have an undefined file.

Table 27 on page 243, Table 28 on page 246, and Table 29 on page 259 show the statements allowed for entry-sequenced data sets, indexed data sets, and relative record data sets, respectively.

```
              SEQUENTIAL        KEYED SEQUENTIAL      DIRECT

INPUT    ESDS              ESDS                 KSDS
         KSDS              KSDS                 RRDS
         RRDS              RRDS                 Path(U)
         Path(N)           Path(N)
         Path(U)           Path(U)

OUTPUT   ESDS              ESDS                 KSDS
         RRDS              KSDS                 RRDS
                           RRDS                 Path(U)

UPDATE   ESDS              ESDS                 KSDS
         KSDS              KSDS                 RRDS
         RRDS              RRDS                 Path(U)
         Path(N)           Path(N)
         Path(U)           Path(U)


Key:  ESDS      Entry-sequenced data set
      KSDS      Key-sequenced data set
      RRDS      Relative record data set
      Path(N)   Alternate index path with nonunique keys
      Path(U)   Alternate index path with unique keys

You can combine the attributes on the left with those at
the top of the figure for the data sets and paths shown.
For example, only an ESDS and an RRDS can be SEQUENTIAL OUTPUT.

PL/I does not support dummy VSAM data sets.
```

Figure 34. VSAM data sets and allowed file attributes

Table 26. Processing Allowed on Alternate Index Paths

Base Cluster Type	Alternate Index Key Type	Processing	Restrictions
KSDS	Unique key	As normal KSDS	May not modify key of access.
	Nonunique key	Limited keyed access	
			May not modify key of access.
ESDS	Unique key	As KSDS	No deletion.
	Nonunique key	Limited keyed access	May not modify key of access.
			No deletion.
			May not modify key of access.

Defining files for VSAM data sets

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
           [KEYED]
             ENVIRONMENT(options);
```

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
           [KEYED]
             ENVIRONMENT(options);
```

Table 13 on page 178 shows the default attributes. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Some combinations of the file attributes INPUT or OUTPUT or UPDATE and DIRECT or SEQUENTIAL or KEYED SEQUENTIAL are allowed only for certain types of VSAM data sets. Figure 34 on page 238 shows the compatible combinations.

Specifying ENVIRONMENT options

Many of the options of the ENVIRONMENT attribute affecting data set structure are not needed for VSAM data sets. If you specify them, they are either ignored or 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 VSAM data sets are:

```
BKWD
BUFND (n)
BUFNI (n)
BUFSP (n)
GENKEY
PASSWORD (password-specification)
REUSE
SCALARVARYING
SKIP
VSAM
```

GENKEY and SCALARVARYING options have the same effect as they do when you use them for non-VSAM data sets. Note that under VSAM RLS, options BUFND, BUFNI and BUFSP are ignored.

The options that are checked for a VSAM data set are RECSIZE and, for a key-sequenced data set, KEYLENGTH and KEYLOC. Table 13 on page 178 shows which options are ignored for VSAM. Table 13 on page 178 also shows the required and default options.

For VSAM data sets, you specify the maximum and average lengths of the records to the Access Method Services utility when you define the data set. If you include the RECSIZE option in the file declaration for checking purposes, specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

BKWD option

Use the BKWD option to specify backward processing for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file associated with a VSAM data set.

►►──BKWD──►◄

Sequential reads (that is, reads without the KEY option) retrieve the previous record in sequence. For indexed data sets, the previous record is, in general, the record with the next lower key. However, if you are accessing the data set via a nonunique *alternate index*, records with the same key are recovered in their normal sequence. For example, if the records are:

```
A B C1 C2 C3 D E
```

where C1, C2, and C3 have the same key, they are recovered in the sequence:

```
E D C1 C2 C3 B A
```

When a file with the BKWD option is opened, the data set is positioned at the last record. ENDFILE is raised in the normal way when the start of the data set is reached.

Do not specify the BKWD option with either the REUSE option or the GENKEY option. Also, the WRITE statement is not allowed for files declared with the BKWD option.

BUFND option

Use the BUFND option to specify the number of data buffers required for a VSAM data set.

►►──BUFND──(*n*)──◄◄

n   specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

Multiple data buffers help performance when the file has the SEQUENTIAL attribute and you are processing long group of contiguous records sequentially.

BUFNI option

Use the BUFNI option to specify the number of index buffers required for a VSAM key-sequence data set.

►►──BUFNI──(*n*)──◄◄

n   specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

Multiple index buffers help performance when the file has the KEYED attribute. Specify at least as many index buffers as there are levels in the index.

BUFSP option

Use the BUFSP option to specify, in bytes, the total buffer space required for a VSAM data set (for both the data and index components).

►►──BUFSP──(*n*)──◄◄

n   specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

It is usually preferable to specify BUFNI and BUFND options rather than BUFSP.

GENKEY option

For the description of this option, see "GENKEY option — key classification" on page 183.

PASSWORD option

When you define a VSAM data set to the system (using the DEFINE command of Access Method Services), you can associate READ and UPDATE passwords with it. From that point on, you must include the appropriate password in the declaration of any PL/I file that you use to access the data set.

►►──PASSWORD──(──*password-specification*──)─────────────────────────────►◄

password-specification
> is a character constant or character variable that specifies the password for the type of access your program requires. If you specify a constant, it must not contain a repetition factor; if you specify a variable, it must be level-1, element, static, and unsubscripted.

The character string is padded or truncated to 8 characters and passed to VSAM for inspection. If the password is incorrect, the system operator is given a number of chances to specify the correct password. You specify the number of chances to be allowed when you define the data set. After this number of unsuccessful tries, the UNDEFINEDFILE condition is raised.

REUSE option

Use the REUSE option to specify that an OUTPUT file associated with a VSAM data set is to be used as a work file.

►►──REUSE──►◄

The data set is treated as an empty data set each time the file is opened. Any secondary allocations for the data set are released, and the data set is treated exactly as if it were being opened for the first time.

Do not associate a file that has the REUSE option with a data set that has *alternate indexes* or the BKWD option, and do not open it for INPUT or UPDATE.

The REUSE option takes effect only if you specify REUSE in the Access Method Services DEFINE CLUSTER command.

SKIP option

Use the SKIP option of the ENVIRONMENT attribute to specify that the VSAM OPTCD ″SKP″ is to be used whenever possible. It is applicable to key-sequenced data sets that you access by means of a KEYED SEQUENTIAL INPUT or UPDATE file.

►►──SKIP───►◄

You should specify this option for the file if your program accesses individual records scattered throughout the data set, but does so primarily in ascending key order.

Omit this option if your program reads large numbers of records sequentially without the use of the KEY option, or if it inserts large numbers of records at specific points in the data set (mass sequential insert).

It is never an error to specify (or omit) the SKIP option; its effect on performance is significant only in the circumstances described.

VSAM option

You must specify the VSAM option for VSAM data sets.

►►──VSAM───►◄

Performance options

You can specify the buffer options in the AMP parameter of the DD statement; they are explained in your Access Method Services manual.

Defining Files for Alternate Index Paths

VSAM allows you to define alternate indexes on key sequenced and entry sequenced data sets. This enables you to access key sequenced data sets in a number of ways other than from the prime index. This also allows you to index and access entry sequenced data sets by key or sequentially in order of the keys. Consequently, data created in one form can be accessed in a large number of different ways. For example, an employee file might be indexed by personnel number, by name, and also by department number.

When an alternate index has been built, you actually access the data set through a third object known as an alternate index *path* that acts as a connection between the alternate index and the data set.

Two types of alternate indexes are allowed—unique key and nonunique key. For a unique key alternate index, each record must have a different alternate key. For a nonunique key alternate index, any number of records can have the same alternate key. In the example suggested above, the alternate index using the names could be a unique key alternate index (provided each person had a different name). The alternate index using the department number would be a nonunique key alternate index because more than one person would be in each department.

In most respects, you can treat a data set accessed through a unique key alternate index path like a KSDS accessed through its prime index. You can access the records by key or sequentially, you can update records, and you can add new records. If the data set is a KSDS, you can delete records, and alter the length of updated records. Restrictions and allowed processing are shown in Table 26 on page 238. When you add or delete records, all indexes associated with the data set are by default altered to reflect the new situation.

In data sets accessed through a nonunique key alternate index path, the record accessed is determined by the key and the sequence. The key can be used to establish positioning so that sequential access can follow. The use of the key accesses the first record with that key. When the data set is read backwards, only the order of the keys is reversed. The order of the records with the same key remains the same whichever way the data set is read.

Defining VSAM data sets

Use the DEFINE CLUSTER command of Access Method Services to define and catalog VSAM data sets. To use the DEFINE command, you need to know:

- The name and password of the master catalog if the master catalog is password protected
- The name and password of the VSAM private catalog you are using if you are not using the master catalog
- Whether VSAM space for your data set is available
- The type of VSAM data set you are going to create
- The volume on which your data set is to be placed
- The average and maximum record size in your data set
- The position and length of the key for an indexed data set
- The space to be allocated for your data set
- How to code the DEFINE command
- How to use the Access Method Services program.

When you have the information, you are in a position to code the DEFINE command and then define and catalog the data set using Access Method Services.

Entry-sequenced data sets

The statements and options allowed for files associated with an ESDS are shown in Table 27.

Table 27. Statements and options allowed for loading and accessing VSAM entry-sequenced data sets

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

Table 27. Statements and options allowed for loading and accessing VSAM entry-sequenced data sets (continued)

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

Notes:

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

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

3. The expression used in the KEY option must be a relative byte address, previously obtained by means of the KEYTO option.

Loading an ESDS

When an ESDS is being loaded, the associated file must be opened for SEQUENTIAL OUTPUT. The records are retained in the order in which they are presented.

You can use the KEYTO option to obtain the relative byte address of each record as it is written. You can subsequently use these keys to achieve keyed access to the data set.

Using a SEQUENTIAL file to access an ESDS

You can open a SEQUENTIAL file that is used to access an ESDS with either the INPUT or the UPDATE attribute. If you use either of the options KEY or KEYTO, the file must also have the KEYED attribute.

Sequential access is in the order that the records were originally loaded into the data set. You can use the KEYTO option on the READ statements to recover the RBAs of the records that are read. If you use the KEY option, the record that is recovered is the one with the RBA you specify. Subsequent sequential access continues from the new position in the data set.

For an UPDATE file, the WRITE statement adds a new record at the end of the data set. With a REWRITE statement, the record rewritten is the one with the specified RBA if you use the KEY option; otherwise, it is the record accessed on the previous READ. You must not attempt to change the length of the record that is being replaced with a REWRITE statement.

The DELETE statement is not allowed for entry-sequenced data sets.

Defining and loading an ESDS

In Figure 35 on page 245, the data set is defined with the DEFINE CLUSTER command and given the name PLIVSAM.AJC1.BASE. The NONINDEXED keyword causes an ESDS to be defined.

The PL/I program writes the data set using a SEQUENTIAL OUTPUT file and a WRITE FROM statement. The DD statement for the file contains the DSNAME of the data set given in the NAME parameter of the DEFINE CLUSTER command.

The RBA 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);
WRITE FILE(FAMFILE) FROM (STRING)
 KEYTO(CHARS);
```

Note that the keys would not normally be printable, but could be retained for subsequent use.

The cataloged procedure IBMZCBG is used. Because the same program (in Figure 35) can be used for adding records to the data set, it is retained in a library. This procedure is shown in the next example.

```
//OPT9#7  JOB
//STEP1    EXEC   PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD    *
     DEFINE CLUSTER -
      (NAME(PLIVSAM.AJC1.BASE) -
      VOLUMES(nnnnnn) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2))
/*
//STEP2  EXEC IBMZCLG
//PLI.SYSIN      DD *
   CREATE: PROC OPTIONS(MAIN);

      DCL
        FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
        IN FILE RECORD INPUT,
        STRING CHAR(80),
        EOF BIT(1) INIT('0'B);

      ON ENDFILE(IN) EOF='1'B;

      READ FILE(IN) INTO (STRING);
      DO I=1 BY 1 WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
        WRITE FILE(FAMFILE) FROM (STRING);
        READ FILE(IN) INTO (STRING);
      END;

      PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
    END;
/*
//LKED.SYSLMOD  DD  DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
//        UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.FAMFILE DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=OLD
//GO.IN      DD     *
FRED               69           M
ANDY               70           M
SUZAN              72           F
/*
```

Figure 35. Defining and loading an entry-sequenced data set (ESDS)

Updating an ESDS

Figure 36 shows the addition of a new record on the end of an ESDS. This is done by executing again the program shown in Figure 35 on page 245. A SEQUENTIAL OUTPUT file is used and the data set associated with it by use of the DSNAME parameter specifying the name PLIVSAM.AJC1.BASE specified in the DEFINE command shown in Figure 35 on page 245.

```
//OPT9#8    JOB
//STEP1     EXEC  PGM=PGMA
//STEPLIB   DD    DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP)
//          DD    DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT  DD    SYSOUT=A
//FAMFILE   DD    DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//IN        DD    *
JANE                    75              F
//
```

Figure 36. Updating an ESDS

You can rewrite existing records in an ESDS, provided that the length of the record is not changed. You can use a SEQUENTIAL or KEYED SEQUENTIAL update file to do this. If you use keys, they can be the RBAs or keys of an *alternate index* path.

Delete is not allowed for ESDS.

Key-sequenced and indexed entry-sequenced data sets

The statements and options allowed for indexed VSAM data sets are shown in Table 28. An indexed data set can be a KSDS with its prime index, or either a KSDS or an ESDS with an *alternate index* Except where otherwise stated, the following description applies to all indexed VSAM data sets.

Table 28. Statements and options allowed for loading and accessing VSAM indexed data sets

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

Table 28. Statements and options allowed for loading and accessing VSAM indexed data sets (continued)

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

Table 28. Statements and options allowed for loading and accessing VSAM indexed data sets (continued)

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

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the declaration.

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

3. Do not associate a SEQUENTIAL OUTPUT file with a data set accessed via an *alternate index.*

4. Do not associate a DIRECT file with a data set accessed via a nonunique *alternate index.*

5. DELETE statements are not allowed for a file associated with an ESDS accessed via an *alternate index.*

Loading a KSDS or indexed ESDS: When a KSDS is being loaded, you must open the associated file for KEYED SEQUENTIAL OUTPUT. You must present the records in ascending key order, and you must use the KEYFROM option. Note that you must use the prime index for loading the data set; you cannot load a VSAM data set via an *alternate index.*

If a KSDS already contains some records, and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can add only records at the end of the data set. The rules given in the previous paragraph apply; in particular, the first record you present must have a key greater than the highest key present on the data set.

Figure 37 on page 249 shows the DEFINE command used to define a KSDS. The data set is given the name PLIVSAM.AJC2.BASE and defined as a KSDS because of the use of the INDEXED operand. The position of the keys within the record is defined in the KEYS operand.

Within the PL/I program, a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A KSDS must be loaded in this manner.

The file is associated with the data set by a DD statement which uses the name given in the DEFINE command as the DSNAME parameter.

```
//OPT9#12  JOB
//    EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT  DD  SYSOUT=A
//SYSIN     DD  *
   DEFINE CLUSTER -
      (NAME(PLIVSAM.AJC2.BASE) -
      VOLUMES(nnnnnn) -
      INDEXED -
      TRACKS(3 1) -
      KEYS(20 0) -
      RECORDSIZE(23 80))
/*
//    EXEC IBMZCBG
//PLI.SYSIN       DD *
 TELNOS: PROC OPTIONS(MAIN);

        DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD POS(1),
            NUMBER CHAR(3) DEF CARD POS(21),
            OUTREC CHAR(23) DEF CARD POS(1),
            EOF BIT(1) INIT('0'B);

        ON ENDFILE(SYSIN) EOF='1'B;

        OPEN FILE(DIREC) OUTPUT;

        GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
        WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;

       CLOSE FILE(DIREC);

       END TELNOS;
/*
//GO.DIREC  DD  DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN  DD  *
ACTION,G.          162
BAKER,R.           152
BRAMLEY,O.H.       248
CHEESEMAN,D.       141
CORY,G.            336
ELLIOTT,D.         875
FIGGINS,S.         413
HARVEY,C.D.W.      205
HASTINGS,G.M.      391
KENDALL,J.G.       294
LANCASTER,W.R.     624
MILES,R.           233
NEWMAN,M.W.        450
PITT,W.H.          515
ROLF,D.E.          114
SHEERS,C.D.        241
SUTCLIFFE,M.       472
TAYLOR,G.C.        407
WILTON,L.W.        404
WINSTONE,E.M.      307
//
```

Figure 37. Defining and loading a key-sequenced data set (KSDS)

Using a SEQUENTIAL file to access a KSDS or indexed ESDS: You can open a
SEQUENTIAL file that is used to access a KSDS with either the INPUT or the
UPDATE attribute.

For READ statements without the KEY option, the records are recovered in
ascending key order (or in descending key order if the BKWD option is used). You
can obtain the key of a record recovered in this way by means of the KEYTO
option.

If you use the KEY option, the record recovered by a READ statement is the one with the specified key. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, without respect to the position of any previous access. If you are accessing the data set via a unique index, the KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists on the data set. For a nonunique index, subsequent retrieval of records with the same key is in the order that they were added to the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the first record with the specified key; otherwise, it is the record that was accessed by the previous READ statement. When you rewrite a record using an *alternate index*, do not change the prime key of the record.

Using a DIRECT file to access a KSDS or indexed ESDS: You can open a DIRECT file that is used to access an indexed VSAM data set with the INPUT, OUTPUT, or UPDATE attribute. Do not use a DIRECT file to access the data set via a nonunique index.

If you use a DIRECT OUTPUT file to add records to the data set, and if an attempt is made to insert a record with the same key as a record that already exists, the KEY condition is raised.

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

Figure 38 on page 251 shows one method by which a KSDS can be updated using the prime index.

```
//OPT9#13  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 DIRUPDT: PROC OPTIONS(MAIN);

        DCL DIREC FILE RECORD KEYED ENV(VSAM),
            ONCODE BUILTIN,
            OUTREC CHAR(23),
            NUMBER CHAR(3) DEF OUTREC POS(21),
            NAME CHAR(20) DEF OUTREC,
            CODE CHAR(1),
            EOF BIT(1) INIT('0'B);

        ON ENDFILE(SYSIN) EOF='1'B;

        ON KEY(DIREC) BEGIN;
         IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                          ('NOT FOUND: ',NAME)(A(15),A);
         IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                        ('DUPLICATE: ',NAME)(A(15),A);

         END;

        OPEN FILE(DIREC) DIRECT UPDATE;

      GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
        (COLUMN(1),A(20),A(3),A(1));
      DO WHILE (¬EOF);
      PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
      (A(1),A(20),A(1),A(3),A(1),A(1));
      SELECT (CODE);
         WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
         WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
         WHEN('D') DELETE FILE(DIREC) KEY(NAME);
         OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME) (A(15),A);
      END;
      GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
        (COLUMN(1),A(20),A(3),A(1));
      END;

        CLOSE FILE(DIREC);
        PUT FILE(SYSPRINT) PAGE;
        OPEN FILE(DIREC) SEQUENTIAL INPUT;

        EOF='0'B;
        ON ENDFILE(DIREC) EOF='1'B;

        READ FILE(DIREC) INTO(OUTREC);
        DO WHILE(¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
        READ FILE(DIREC) INTO(OUTREC);
        END;
        CLOSE FILE(DIREC);
   END DIRUPDT;
```

Figure 38. Updating a KSDS (Part 1 of 2)

```
/*
//GO.DIREC DD DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.         516C
GOODFELLOW,D.T.     889A
MILES,R.             D
HARVEY,C.D.W.       209A
BARTLETT,S.G.       183A
CORY,G.              D
READ,K.M.           001A
PITT,W.H.
ROLF,D.F.            D
ELLIOTT,D.          291C
HASTINGS,G.M.        D
BRAMLEY,O.H.        439C
/*
```

Figure 38. Updating a KSDS (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

At the label NEXT, the name, number, and code are read in and action taken according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished (at the label PRINT), the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

The file is associated with the data set by a DD statement that uses the DSNAME PLIVSAM.AJC2.BASE defined in the Access Method Services DEFINE CLUSTER command in Figure 37 on page 249.

Methods of updating a KSDS: There are a number of methods of updating a KSDS. The method shown using a DIRECT file is suitable for the data as it is shown in the example. For mass sequential insertion, use a KEYED SEQUENTIAL UPDATE file. This gives faster performance because the data is written onto the data set only when strictly necessary and not after every write statement, and because the balance of free space within the data set is retained.

Statements to achieve effective mass sequential insertion are:
```
DCL DIREC KEYED SEQUENTIAL UPDATE
    ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
 KEYFROM(NAME);
```

The PL/I input/output routines detect that the keys are in sequence and make the correct requests to VSAM. If the keys are not in sequence, this too is detected and no error occurs, although the performance advantage is lost.

Alternate Indexes for KSDSs or Indexed ESDSs

Alternate indexes allow you to access KSDSs or indexed ESDSs in various ways, using either unique or nonunique keys.

Unique Key Alternate Index Path: Figure 39 shows the creation of a unique key alternate index path for the ESDS defined and loaded in Figure 35 on page 245. Using this path, the data set is indexed by the name of the child in the first 15 bytes of the record.

Three Access Method Services commands are used. These are:

DEFINE ALTERNATEINDEX: defines the alternate index as a data set to VSAM.

BLDINDEX: places the pointers to the relevant records in the alternate index.

DEFINE PATH: defines an entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE operands of BLDINDEX and for the sort files. Care should be taken that the correct names are specified at the various points.

```
//OPT9#9    JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD   SYSOUT=A
//SYSIN     DD   *
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC1.ALPHIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 1) -
     KEYS(15 0) -
     RECORDSIZE(20 40) -
     UNIQUEKEY -
   RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC1.ALPHIND,DISP=SHR
//SYSPRINT DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC1.ALPHPATH) -
     PATHENTRY(PLIVSAM.AJC1.ALPHIND))
//
```

Figure 39. Creating a Unique Key Alternate Index Path for an ESDS

Nonunique Key Alternate Index Path: Figure 40 on page 254 shows the creation of a nonunique key alternate index path for an ESDS. The alternate index enables the data to be selected by the sex of the children. This enables the girls or the boys to be accessed separately and every member of each group to be accessed by use of the key.

The three Access Method Services commands used are:

DEFINE ALTERNATEINDEX: defines the alternate index as a data set to VSAM.

BLDINDEX: places the pointers to the relevant records in the alternate index.

DEFINE PATH: defines an entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE operands of BLDINDEX and for the sort files. Care should be taken that the correct names are specified at the various points.

The fact that the index has nonunique keys is specified by the use of the
NONUNIQUEKEY operand. When creating an index with nonunique keys, be
careful to ensure that the RECORDSIZE you specify is large enough. In a
nonunique alternate index, each alternate index record contains pointers to all the
records that have the associated index key. The pointer takes the form of an RBA
for an ESDS and the prime key for a KSDS. When a large number of records might
have the same key, a large record is required.

```
//OPT9#10   JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   /* care must be taken with recordsize */
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC1.SEXIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 1) -
     KEYS(1 37) -
     RECORDSIZE(20 400) -
     NONUNIQUEKEY -
    RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC1.SEXIND,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC1.SEXPATH) -
     PATHENTRY(PLIVSAM.AJC1.SEXIND))
//
```

Figure 40. Creating a Nonunique Key Alternate Index Path for an ESDS

Figure 41 on page 255 shows the creation of a unique key alternate index path for
a KSDS. The data set is indexed by the telephone number, enabling the number to
be used as a key to discover the name of the person on that extension. The fact
that keys are to be unique is specified by UNIQUEKEY. Also, the data set will be
able to be listed in numerical order to show which numbers are not used. The
three Access Method Services commands used are:

DEFINE ALTERNATEINDEX: defines the data set that will hold the alternate
index data.

BLDINDEX: places the pointer to the relevant records in the alternate index.

DEFINE PATH: defines the entity that can be associated with a PL/I file in a DD
statement.

DD statements are required for the INFILE and OUTFILE of BLDINDEX and for
the sort files. Be careful not to confuse the names involved.

```
//OPT9#14   JOB
//STEP1     EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD   SYSOUT=A
//SYSIN     DD   *
   DEFINE ALTERNATEINDEX -
     (NAME(PLIVSAM.AJC2.NUMIND) -
     VOLUMES(nnnnnn) -
     TRACKS(4 4) -
     KEYS(3 20) -
     RECORDSIZE(24 48) -
     UNIQUEKEY -
    RELATE(PLIVSAM.AJC2.BASE))
/*
//STEP2     EXEC PGM=IDCAMS,REGION=512K
//DD1       DD   DSNAME=PLIVSAM.AJC2.BASE,DISP=SHR
//DD2       DD   DSNAME=PLIVSAM.AJC2.NUMIND,DISP=SHR
//SYSPRINT DD   SYSOUT=A
//SYSIN     DD   *
   BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
   DEFINE  PATH -
     (NAME(PLIVSAM.AJC2.NUMPATH) -
     PATHENTRY(PLIVSAM.AJC2.NUMIND))
//
```

Figure 41. Creating a unique Key Alternate Index Path for a KSDS

When creating an alternate index with a unique key, you should ensure that no further records could be included with the same alternate key. In practice, a unique key alternate index would not be entirely satisfactory for a telephone directory as it would not allow two people to have the same number. Similarly, the prime key would prevent one person having two numbers. A solution would be to have an ESDS with two nonunique key alternate indexes, or to restructure the data format to allow more than one number per person and to have a nonunique key alternate index for the numbers.

Detecting Nonunique Alternate Index Keys: If you are accessing a VSAM data set by means of an alternate index path, the presence of nonunique keys can be detected by means of the SAMEKEY built-in function. After each retrieval, SAMEKEY indicates whether any further records exist with the same alternate index key as the record just retrieved. Hence, it is possible to stop at the last of a series of records with nonunique keys without having to read beyond the last record. SAMEKEY (file-reference) returns '1'B if the input/output statement has completed successfully and the accessed record is followed by another with the same key; otherwise, it returns '0'B.

Using Alternate Indexes with ESDSs: Figure 42 on page 257 shows the use of alternate indexes and backward reading on an ESDS. The program has four files:

BASEFLE

reads the base data set forward.

BACKFLE

reads the base data set backward.

ALPHFLE

is the alphabetic alternate index path indexing the children by name.

SEXFILE

is the alternate index path that corresponds to the sex of the children.

There are DD statements for all the files. They connect BASEFLE and BACKFLE to the base data set by specifying the name of the base data set in the DSNAME parameter, and connect ALPHFLE and SEXFILE by specifying the names of the paths given in Figure 39 on page 253 and Figure 40 on page 254.

The program uses SEQUENTIAL files to access the data and print it first in the normal order, then in the reverse order. At the label AGEQUERY, a DIRECT file is used to read the data associated with an alternate index key in the unique alternate index.

Finally, at the label SPRINT, a KEYED SEQUENTIAL file is used to print a list of the females in the family, using the nonunique key alternate index path. The SAMEKEY built-in function is used to read all the records with the same key. The names of the females will be accessed in the order in which they were originally entered. This will happen whether the file is read forward or backward. For a nonunique key path, the BKWD option only affects the order in which the keys are read; the order of items with the same key remains the same as it is when the file is read forward.

Deletion: At the end of the example, the Access Method Services DELETE command is used to delete the base data set. When this is done, the associated alternate indexes and paths will also be deleted.

Using Alternate Indexes with KSDSs: Figure 43 on page 259 shows the use of a path with a unique alternate index key to update a KSDS and then to access and print it in the order of the alternate index.

The alternate index path is associated with the PL/I file by a DD statement that specifies the name of the path (PLIVSAM.AJC2.NUMPATH, given in the DEFINE PATH command in Figure 41 on page 255) as the DSNAME.

In the first section of the program, a DIRECT OUTPUT file is used to insert a new record using the alternate index key. Note that any alteration made with an alternate index must not alter the prime key or the alternate index key of access of an existing record. Also, the alternation must not add a duplicate key in the prime index or any unique key alternate index.

In the second section of the program (at the label PRINTIT), the data set is read in the order of the alternate index keys using a SEQUENTIAL INPUT file. It is then printed onto SYSPRINT.

```
//OPT9#15  JOB
//STEP1  EXEC IBMZCLG
//PLI.SYSIN      DD *
  READIT: PROC OPTIONS(MAIN);
    DCL   BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
                 /*File to read base data set forward */
          BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
                 /*File to read base data set backward */
          ALPHFLE FILE DIRECT INPUT ENV(VSAM),
          /*File to access via unique alternate index path */
          SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
           /*File to access via nonunique alternate index path */
          STRING CHAR(80),  /*String to be read into */
          1 STRUC DEF (STRING),
            2 NAME CHAR(25),
            2 DATE_OF_BIRTH CHAR(2),
            2 FILL CHAR(10),
            2 SEX CHAR(1);
          DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;
          DCL EOF BIT(1) INIT('0'B);

       /*Print out the family eldest first*/

       ON ENDFILE(BASEFLE) EOF='1'B;
       PUT EDIT('FAMILY ELDEST FIRST')(A);
       READ FILE(BASEFLE) INTO (STRING);
       DO WHILE(¬EOF);
          PUT SKIP EDIT(STRING)(A);
          READ FILE(BASEFLE) INTO (STRING);
       END;
       CLOSE FILE(BASEFLE);
       PUT SKIP(2);
        /*Close before using data set from other file not
          necessary but good practice to prevent potential
          problems*/

       EOF='0'B;
       ON ENDFILE(BACKFLE) EOF='1'B;
       PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);
       READ FILE(BACKFLE) INTO(STRING);
       DO WHILE(¬EOF);
         PUT SKIP EDIT(STRING)(A);
         READ FILE(BACKFLE) INTO (STRING);
       END;

       CLOSE FILE(BACKFLE);
       PUT SKIP(2);

       /*Print date of birth of child specified in the file
         SYSIN*/
       ON KEY(ALPHFLE) BEGIN;
          PUT SKIP EDIT
             (NAMEHOLD,' NOT A MEMBER OF THE SMITH FAMILY') (A);
          GO TO SPRINT;
       END;
```

Figure 42. Alternate Index Paths and Backward Reading with an ESDS (Part 1 of 2)

```
        AGEQUERY:
           EOF='0'B;
           ON ENDFILE(SYSIN) EOF='1'B;
           GET SKIP EDIT(NAMEHOLD)(A(25));
           DO WHILE(¬EOF);
                READ FILE(ALPHFLE) INTO (STRING) KEY(NAMEHOLD);
                PUT SKIP (2) EDIT(NAMEHOLD,' WAS BORN IN ',
                    DATE_OF_BIRTH)(A,X(1),A,X(1),A);
                GET SKIP EDIT(NAMEHOLD)(A(25));
           END;
     SPRINT:
           CLOSE FILE(ALPHFLE);
           PUT SKIP(1);

          /*Use the alternate index to print out all the females in the
            family*/
                ON ENDFILE(SEXFILE) GOTO FINITO;
                PUT SKIP(2) EDIT('ALL THE FEMALES')(A);
                READ FILE(SEXFILE) INTO (STRING) KEY('F');
                PUT SKIP EDIT(STRING)(A);
                DO WHILE(SAMEKEY(SEXFILE));
                   READ FILE(SEXFILE) INTO (STRING);
                   PUT SKIP EDIT(STRING)(A);
                END;

      FINITO:
           END;
     /*
     //GO.BASEFLE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
     //GO.BACKFLE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
     //GO.ALPHFLE   DD   DSN=PLIVSAM.AJC1.ALPHPATH,DISP=SHR
     //GO.SEXFILE   DD   DSN=PLIVSAM.AJC1.SEXPATH,DISP=SHR
     //GO.SYSIN     DD   *
     ANDY
     /*
     //STEP2    EXEC PGM=IDCAMS,REGION=512K
     //SYSPRINT    DD    SYSOUT=A
     //SYSIN       DD    *
          DELETE -
               PLIVSAM.AJC1.BASE
     //
```

Figure 42. Alternate Index Paths and Backward Reading with an ESDS (Part 2 of 2)

```
//OPT9#16   JOB
//STEP1     EXEC IBMZCLG,REGION.GO=256K
//PLI.SYSIN DD   *
   ALTER: PROC OPTIONS(MAIN);
     DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
         NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
         IN FILE RECORD,
         STRING CHAR(80),
         NAME CHAR(20) DEF STRING,
         NUMBER CHAR(3) DEF STRING POS(21),
         DATA CHAR(23) DEF STRING,
         EOF BIT(1) INIT('0'B);

     ON KEY (NUMFLE1) BEGIN;
       PUT SKIP EDIT('DUPLICATE NUMBER')(A);
     END;

     ON ENDFILE(IN) EOF='1'B;

     READ FILE(IN) INTO (STRING);
     DO WHILE(¬EOF);
       PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
       WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
       READ FILE(IN) INTO (STRING);
     END;

     CLOSE FILE(NUMFLE1);

     EOF='0'B;
     ON ENDFILE(NUMFLE2) EOF='1'B;

     READ FILE(NUMFLE2) INTO (STRING);
     DO WHILE(¬EOF);
       PUT SKIP EDIT(DATA)(A);
       READ FILE(NUMFLE2) INTO (STRING);
     END;

     PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
   END;
/*
//GO.IN    DD   *
RIERA L           123
/*
//NUMFLE1   DD   DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//NUMFLE2   DD   DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//STEP2    EXEC PGM=IDCAMS,COND=EVEN
//SYSPRINT  DD   SYSOUT=A
//SYSIN   DD   *
    DELETE -
    PLIVSAM.AJC2.BASE
//
```

Figure 43. Using a Unique Alternate Index Path to Access a KSDS

Relative-record data sets

The statements and options allowed for VSAM relative-record data sets (RRDS) are shown in Table 29.

Table 29. Statements and options allowed for loading and accessing VSAM relative-record data sets

File declaration[1]	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)

Table 29. Statements and options allowed for loading and accessing VSAM relative-record data sets (continued)

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

Table 29. Statements and options allowed for loading and accessing VSAM relative-record data sets (continued)

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

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, your declaration must also include the attribute KEYED.

   The UNLOCK statement for DIRECT UPDATE files is ignored if you use it for files associated with a VSAM RRDS.

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

Loading an RRDS: When an RRDS is being loaded, you must open the associated file for OUTPUT. Use either a DIRECT or a SEQUENTIAL file.

For a DIRECT OUTPUT file, each record is placed in the position specified by the relative record number (or key) in the KEYFROM option of the WRITE statement (see "Keys for VSAM data sets" on page 236).

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 means of the KEYTO option.

If you want to load an RRDS sequentially, without use of the KEYFROM or KEYTO options, your file is not required to have the KEYED attribute.

It is an error to attempt to load a record into a position that already contains a record: if you use the KEYFROM option, the KEY condition is raised; if you omit it, the ERROR condition is raised.

In Figure 44 on page 262, the data set is defined with a DEFINE CLUSTER command and given the name PLIVSAM.AJC3.BASE. The fact that it is an RRDS is determined by the NUMBERED keyword. In the PL/I program, it is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

If the data had been in order and the keys in sequence, it would have been possible to use a SEQUENTIAL file and write into the data set from the start. The records would then have been placed in the next available slot and given the appropriate number. The number of the key for each record could have been returned using the KEYTO option.

The PL/I file is associated with the data set by the DD statement, which uses as the DSNAME the name given in the DEFINE CLUSTER command.

```
//OPT9#17  JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD *
        DEFINE CLUSTER -
            (NAME(PLIVSAM.AJC3.BASE) -
            VOLUMES(nnnnnn) -
            NUMBERED -
            TRACKS(2 2) -
            RECORDSIZE(20 20))
/*
//STEP2  EXEC IBMZCBG
//PLI.SYSIN      DD *
 CRR1:   PROC OPTIONS(MAIN);
        DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD,
            NUMBER CHAR(2) DEF CARD POS(21),
            IOFIELD CHAR(20),
            EOF BIT(1) INIT('0'B);
        ON ENDFILE (SYSIN) EOF='1'B;
        OPEN FILE(NOS);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
       DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
        IOFIELD=NAME;
        WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;
        CLOSE FILE(NOS);
 END CRR1;
/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.       28
CHEESNAME,L.       11
CORY,G.            36
ELLIOTT,D.         85
FIGGINS.E.S.       43
HARVEY,C.D.W.      25
HASTINGS,G.M.      31
KENDALL,J.G.       24
LANCASTER,W.R.     64
MILES,R.           23
NEWMAN,M.W.        40
PITT,W.H.          55
ROLF,D.E.          14
SHEERS,C.D.        21
SURCLIFFE,M.       42
TAYLOR,G.C.        47
WILTON,L.W.        44
WINSTONE,E.M.      37
//
```

Figure 44. Defining and loading a relative-record data set (RRDS)

Using a SEQUENTIAL file to access an RRDS: You can open a SEQUENTIAL
file that is used to access an RRDS with either the INPUT or the UPDATE
attribute. If you use any of the options KEY, KEYTO, or KEYFROM, your file must
also have the KEYED attribute.

For READ statements without the KEY option, the records are recovered in
ascending relative record number order. Any empty slots in the data set are
skipped.

If you use the KEY option, the record recovered by a READ statement is the one with the relative record number you specify. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with or without the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, regardless of the position of any previous access. For WRITE with the KEYFROM option, the KEY condition is raised if an attempt is made to insert a record with the same relative record number as a record that already exists on the data set. If you omit the KEYFROM option, an attempt is made to write the record in the next slot, relative to the current position. The ERROR condition is raised if this slot is not empty.

You can use the KEYTO option to recover the key of a record that is added by means of a WRITE statement without the KEYFROM option.

REWRITE statements, with or without the KEY option, are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the record with the relative record number you specify; otherwise, it is the record that was accessed by the previous READ statement.

DELETE statements, with or without the KEY option, can be used to delete records from the dataset.

Using a DIRECT file to access an RRDS: A DIRECT file used to access an RRDS can have the OUTPUT, INPUT, or UPDATE attribute. You can read, write, rewrite, or delete records exactly as though a KEYED SEQUENTIAL file were used.

Figure 45 on page 264 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under VSAM.

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

The PL/I file is associated with the data sets by a DD statement that specifies the DSNAME PLIVSAM.AJC3.BASE, the name given in the DEFINE CLUSTER command in Figure 45 on page 264.

At the end of the example, the DELETE command is used to delete the data set.

```
//* NOTE: WITH A WRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A DUPLICATE MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO CORRESPONDS TO AN EXISTING NUMBER IN THE LIST,
//*       FOR EXAMPLE, ELLIOT.
//*       WITH A REWRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A NOT FOUND MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO DOES NOT CORRESPOND TO AN EXISTING NUMBER IN
//*       THE LIST, FOR EXAMPLE, NEWMAN AND BRAMLEY.
//OPT9#18  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 ACR1:  PROC OPTIONS(MAIN);
               DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
                 (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
                 BYTE CHAR(1) DEF IOFIELD, EOF BIT(1) INIT('0'B),
                 ONCODE BUILTIN;
        ON ENDFILE(SYSIN) EOF='1'B;
        OPEN FILE(NOS) DIRECT UPDATE;
        ON KEY(NOS) BEGIN;
           IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                   ('NOT FOUND:',NAME)(A(15),A);
           IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                   ('DUPLICATE:',NAME)(A(15),A);
        END;
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
           (A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
        SELECT(CODE);
           WHEN('A') WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           WHEN('C') DO;
            DELETE FILE(NOS) KEY(OLDNO);
            WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           END;
           WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
           OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME)(A(15),A);
        END;
```

Figure 45. Updating an RRDS (Part 1 of 2)

```
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        END;
         CLOSE FILE(NOS);
             PRINT:
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(NOS) SEQUENTIAL INPUT;
         EOF='0'B;
         ON ENDFILE(NOS) EOF='1'B;
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          DO WHILE (¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD)(A(5),A);
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          END;
       CLOSE FILE(NOS);
  END ACR1;

/*
//GO.NOS      DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN    DD  *
NEWMAN,M.W.        5640C
GOODFELLOW,D.T.    89  A
MILES,R.            23D
HARVEY,C.D.W.      29  A
BARTLETT,S.G.      13  A
CORY,G.            36D
READ,K.M.          01  A
PITT,W.H.          55
ROLF,D.F.          14D
ELLIOTT,D.         4285C
HASTINGS,G.M.      31D
BRAMLEY,O.H.       4928C
//STEP3    EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN   DD   *
      DELETE -
          PLIVSAM.AJC3.BASE
//
```

Figure 45. Updating an RRDS (Part 2 of 2)

Using Files Defined for non-VSAM Data Sets

Using Shared Data Sets

PL/I allows cross-region or cross-system sharing of data sets. The support for this type of sharing is provided by VSAM. For details about the support, see the following DFSMS manuals:

- *DFSMS: Using Data Sets*
- *DFSMS: Access Method Services for Catalogs*

Part 3. Improving your program

Chapter 11. Improving performance

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

Selecting compiler options for optimal performance

The compiler options you choose can greatly improve the performance of the code generated by the compiler; however, like most performance considerations, there are trade-offs associated with these choices. Fortunately, you can weigh the trade-offs associated with compiler options without editing your source code because these options can be specified on the command line or in the configuration file.

If you want to avoid details, the least complex way to improve the performance of generated code is to specify the following (nondefault) compiler options:

    OPT(2) or OPT(3)
    DFT(REORDER)

The following sections describe, in more detail, performance improvements and trade-offs associated with specific compiler options.

OPTIMIZE

You can specify the OPTIMIZE option to improve the speed of your program; otherwise, the compiler makes only basic optimization efforts.

Choosing OPTIMIZE(2) directs the compiler to generate code for better performance. Usually, the resultant code is shorter than when the program is compiled under NOOPTIMIZE. Sometimes, however, a longer sequence of instructions runs faster than a shorter sequence. This occurs, for instance, when a branch table is created for a SELECT statement where the values in the WHEN clauses contain gaps. The increased number of instructions generated is usually offset by the execution of fewer instructions in other places.

Choosing OPTIMIZE(3) directs the compiler to generate even better code. However, when you specify the OPTIMIZE(3) option, the compilation will take longer (and sometimes much, much longer) than when you specify OPTIMIZE(2).

GONUMBER

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.

ARCH

Using ARCH(5) allows the compiler to select from the largest set of instructions available under z/OS and thus permits it to generate the most optimal code.

REDUCE

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

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

RULES

Most of the RULES suboptions affect only the severity with which certain coding practices, such as not declaring variables, are flagged and have no impact on performance. However, these suboptions do have an impact on performance.

IBM/ANS

When you use the RULES(IBM) option, the compiler supports scaled FIXED BINARY and, what is more important for performance, generates scaled FIXED BINARY results in some operations. Under RULES(ANS), scaled FIXED BINARY is not supported and scaled FIXED BINARY results are never generated. This means that the code generated under RULES(ANS) always runs at least as fast as the code generated under RULES(IBM), and sometimes runs faster.

For example, consider the following code fragment:

```
dcl (i,j,k) fixed bin(15);
   .
   .
   .
i = j / k;
```

Under RULES(IBM), the result of the division has the attributes FIXED BIN(31,16). This means that a shift instruction is required before the division and several more instructions are needed to perform the assignment.

Under RULES(ANS), the result of the division has the attributes FIXED BIN(15,0). This means that a shift is not needed before the division, and no extra instructions are needed to perform the assignment.

(NO)LAXCTL

Under RULES(LAXCTL), a CONTROLLED variable may be declared with constant extents and yet allocated with different extents.

For instance, under RULES(LAXCTL), you may declare a structure as follows:

```
dcl
  1 a controlled,
    2 b char(17),
    2 c char(29);
```

However, you could then allocate it as follows:

```
allocate
  1 a,
    2 b char(170),
    2 c char(290);
```

This has disastrous consequences for performance because it means that whenever the compiler sees a reference to the structure *A* or to any member of that structure, the compiler is forced to assume that it knows nothing about the lengths, dimensions or offsets of the fields in it.

However, the RULES(NOLAXCTL) option disallows this coding practice: under RULES(NOLAXCTL), if you then want to allocate a CONTROLLED variable with a variable extent, then that extents must be declared either with an asterisk or with a non-constant expression. Consequently, under RULES(NOLAXCTL), when a CONTROLLED variable is declared with constant extents, then the compiler can generate much better code for any reference to that variable.

PREFIX

This option determines if selected PL/I conditions are enabled by default. The default suboptions for PREFIX are set to conform to the PL/I language definition; however, overriding the defaults can have a significant effect on the performance of your program. The default suboptions are:

    CONVERSION
    INVALIDOP
    FIXEDOVERFLOW
    OVERFLOW
    INVALIDOP
    NOSIZE
    NOSTRINGRANGE
    NOSTRINGSIZE
    NOSUBSCRIPTRANGE
    UNDERFLOW
    ZERODIVIDE

By specifying the SIZE, STRINGRANGE, STRINGSIZE, or SUBSCRIPTRANGE suboptions, the compiler generates extra code that helps you pinpoint various problem areas in your source that would otherwise be hard to find. This extra code, however, can slow program performance significantly.

CONVERSION

When you disable the CONVERSION condition, some character-to-numeric conversions are done inline and without checking the validity of the source; therefore, specifying NOCONVERSION also affects program performance.

FIXEDOVERFLOW

On some platforms, the FIXEDOVERFLOW condition is raised by the hardware and the compiler does not need to generate any extra code to detect it.

DEFAULT

Using the DEFAULT option, you can select attribute defaults. As is true with the PREFIX option, the suboptions for DEFAULT are set to conform to the PL/I language definition. Changing the defaults in some instances can affect performance.

Some of the suboptions, such IBM/ANS and ASSIGNABLE/NONASSIGNABLE, have no effect on program performance. But other suboptions can affect performance to varying degrees and, if applied inappropriately, can make your program invalid. The more important of these suboptions are:

BYADDR or BYVALUE

When the DEFAULT(BYADDR) option is in effect, arguments are passed by reference (as required by PL/I) unless an attribute in an entry declaration indicates otherwise. As arguments are passed by reference, the address of the argument is passed from one routine (calling routine) to another (called routine) as the variable itself is passed. Any change made to the argument while in the called routine is reflected in the calling routine when it resumes execution.

Program logic often depends on passing variables by reference. Passing a variable by reference, however, can hinder performance in two ways:
1. Every reference to that parameter requires an extra instruction.
2. 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.

(NO)DESCRIPTOR

The DEFAULT(DESCRIPTOR) option indicates that, by default, a descriptor is passed for any string, area, or aggregate parameter; however, the descriptor is used only if the parameter has nonconstant extents or if the parameter is an array with the NONCONNECTED attribute. In this case, the instructions and space required to pass the descriptor provide no benefit and incur substantial cost (the size of a structure descriptor is often greater than size of the structure itself). Consequently, by specifying DEFAULT(NODESCRIPTOR) and using OPTIONS(DESCRIPTOR) only as needed on PROCEDURE statements and ENTRY declarations, your code runs more optimally.

(NO)INLINE

The 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.

LINKAGE
This suboption tells the compiler the default linkage to use when the LINKAGE suboption of the OPTIONS attribute or option for an entry has not been specified.

The compiler supports various linkages, each with its unique performance characteristics. When you invoke an ENTRY provided by an external entity (such as an operating system), you must use the linkage previously defined for that ENTRY.

As you create your own applications, however, you can choose the linkage convention. The OPTLINK linkage is strongly recommended because it provides significantly better performance than other linkage conventions.

(RE)ORDER
The DEFAULT(ORDER) option indicates that the ORDER option is applied to every block, meaning that variables in that block referenced in ON-units (or blocks dynamically descendant from ON-units) have their latest values. This effectively prohibits almost all optimization on such variables. Consequently, if your program logic allows, use DEFAULT(REORDER) to generate superior code.

NOOVERLAP
The DEFAULT(NOOVERALP) option lets the compiler assume that the source and target in an assignment do not overlap, and it can therefore generate smaller and faster code.

However, if you this option, you must insure that the source and target in assignment do not overlap. For example, under the DEFAULT(NOOVERLAP) option, the assignment in this example would be invalid:

```
dcl c char(20);
substr(c,2,5) = substr(c,1,5);
```

RETURNS(BYVALUE) or RETURNS(BYADDR)
When the DEFAULT(RETURNS(BYVALUE)) option is in effect, the BYVALUE attribute is applied to all RETURNS description lists that do not specify BYADDR. This means that these functions return values in registers, when possible, in order to produce the most optimal code.

Summary of compiler options that improve performance
In summary, the following options (if appropriate for your application) can improve performance:
    OPTIMIZE(3)
    ARCH(5)
    REDUCE
    RULES(ANS NOLAXCTL)
    DEFAULT with the following suboptions

> BYVALUE
> CONNECTED
> NODESCRIPTOR
> INLINE
> LINKAGE(OPTLINK)
> REORDER
> NOOVERLAP
> RETURNS(BYVALUE)

Coding for better performance

As you write code, there is generally more than one correct way to accomplish a given task. Many important factors influence the coding style you choose, including readability and maintainability. The following 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.

GOTO statements

A GOTO statement that uses either a label in another block or a label variable severely limits optimizations that the compiler might perform. If a label array is initialized and declared AUTOMATIC, either implicitly or explicitly, any GOTO to an element of that array will hinder optimization. However, if the array is declared as STATIC, the compiler assumes the CONSTANT attribute for it and no optimization is hindered.

String assignments

When one string is assigned to another, the compiler ensures that:
- The target has the correct value even if the source and target overlap.
- 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.

If a loop control variable is a FIXED BIN, performance is best if it has precision 31 and is SIGNED.

Performance is decreased if your program depends not only on the value of a loop control variable, but also on its address. For example, 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;
```

DESCLOCATOR or DESCLIST

When the DEFAULT(DESCLOCATOR) option is in effect, the compiler passes arguments requiring descriptors (such as strings and structures) via a descriptor locator in much the same way that the old compiler did. More information on descriptors and how they are passed is available in Chapter 20, "PL/I descriptors under CMPAT(LE)," on page 393.

This option allows you to invoke an entry point that is not always passed all of the arguments that it declares.

This option also allows you to continue the somewhat unwise programming practice of passing a structure and receiving it as a pointer.

However, the code generated by the compiler for DEFAULT(DESCLOCATOR) may, in some situations, perform less well than that for DEFAULT(DESCLIST).

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. The compiler does not treat NONASSIGNABLE scalar STATIC variables as true named constants.

The compiler generates better code whenever expressions are evaluated during compilation, so you can use named constants to produce efficient code with no loss in readability. For example, identical object code is produced for the two usages of the VERIFY built-in function in the following example:

```
dcl numeric char  value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );
```

The following examples illustrate how you can use the VALUE attribute to get optimal code without sacrificing readability.

Example with optimal code but no meaningful names
```
dcl  x  bit(8) aligned;

select( x );
  when( '01'b4 )
   .
   .
   .
  when( '02'b4 )
   .
   .
   .
  when( '03'b4 )
   .
   .
   .
end;
```

Example with meaningful names but not optimal code
```
dcl (  a1  init( '01'b4)
      ,a2  init( '02'b4)
      ,a3  init( '03'b4)
      ,a4  init( '04'b4)
```

```
          ,a5  init( '05'b4)
     ) bit(8) aligned static nonassignable;

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
   .
   .
   .
  when( a2 )
   .
   .
   .
  when( a3 )
   .
   .
   .
end;
```

Example with optimal code AND meaningful names

```
dcl (  a1  value( '01'b4)
      ,a2  value( '02'b4)
      ,a3  value( '03'b4)
      ,a4  value( '04'b4)
      ,a5  value( '05'b4)
     ) bit(8);

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
   .
   .
   .
  when( a2 )
   .
   .
   .
  when( a3 )
   .
   .
   .
end;
```

Avoiding calls to library routines

The bitwise operations (prefix NOT, infix AND, infix OR, and infix EXCLUSIVE
OR) are often evaluated by calls to library routines. These operations are, however,
handled without a library call if either of the following conditions is true:
• Both operands are bit(1)
• Both operands are aligned and have the same constant length.

For certain assignments, expressions, and built-in function references, the compiler
generates calls to library routines. If you avoid these calls, your code generally
runs faster.

To help you determine when the compiler generates such calls, the compiler
generates a message whenever a conversion is done using a library routine.

When your code refers to a member of a BASED structure using REFER, the
compiler often has to generate one or more calls to a library routine to map the

structure at run-time. These calls can be expensive, and so when the compiler makes these calls, it will issue a message so that you can locate these potential hot-spots in your code.

If you do have code that uses BASED structures with REFER and which the compiler flags with this message, you may be able to get better performance by passing the structure to a subroutine that declares a corresponding structure with * extents. This will cause the structure to be mapped once at the CALL statement, but there will no further remappings when it is accessed in the called subroutine.

Coding for better performance

Part 4. Using interfaces to other products

Chapter 12. Using the Sort program

The compiler provides an interface called PLISRTx (x = A, B, C, or D) that allows you to make use of the IBM-supplied Sort programs.

To use the Sort program with PLISRTx, you must:

1. Include a call to one of the entry points of PLISRTx, passing it the information on the fields to be sorted. This information includes the length of the records, the maximum amount of storage to use, the name of a variable to be used as a return code, and other information required to carry out the sort.
2. Specify the data sets required by the Sort program in JCL DD statements.

When used from PL/I, the Sort program sorts records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted can be either a data set or a user-written PL/I procedure that the Sort program will call each time a record is required for the sort. Similarly, the destination of the sort can be a data set or a PL/I procedure that handles the sorted records.

Using PL/I procedures allows processing to be done before or after the sort itself, thus allowing a complete sorting operation to be handled completely by a call to the sort interface. It is important to understand that the PL/I procedures handling input or output are called from the Sort program itself and will effectively become part of it.

PL/I can operate with DFSORT™ or a program with the same interface. DFSORT is a release of the program product 5740-SM1. DFSORT has many built-in features you can use to eliminate the need for writing program logic (for example, INCLUDE, OMIT, OUTREC and SUM statement plus the many ICETOOL operators). See *DFSORT Application Programming Guide* for details and *Getting Started with DFSORT* for a tutorial.

The following material applies to DFSORT. Because you can use programs other than DFSORT, the actual capabilities and restrictions vary. For these capabilities and restrictions, see *DFSORT Application Programming Guide*, or the equivalent publication for your sort product.

To use the Sort program you must include the correct PL/I statements in your source program and specify the correct data sets in your JCL.

Preparing to use Sort

Before using Sort, you must determine the type of sort you require, the length and format of the sorting fields in the data, the length of your data records, and the amount of auxiliary and main storage you will allow for sorting.

To determine the PLISRTx entry point that you will use, you must decide the source of your unsorted data, and the destination of your sorted data. You must choose between data sets and PL/I subroutines. Using data sets is simpler to understand and gives faster performance. Using PL/I subroutines gives you more flexibility and more function, enabling you to manipulate or print the data before it

is sorted, and to make immediate use of it in its sorted form. If you decide to use an input or output handling subroutine, you will need to read "Data input and output handling routines" on page 295.

The entry points and the source and destination of data are as follows:

Entry point	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine
PLISRTD	Subroutine	Subroutine

Having determined the entry point you are using, you must now determine the following things about your data set:
- The position of the sorting fields; these can be either the complete record or any part or parts of it
- The type of data these fields represent, for example, character or binary
- Whether you want the sort on each field to be in ascending or descending order
- Whether you want equal records to be retained in the order of the input, or whether their order can be altered during sorting

Specify these options on the SORT statement, which is the first argument to PLISRTx. After you have determined these, you must determine two things about the records to be sorted:
- Whether the record format is fixed or varying
- The length of the record, which is the maximum length for varying

Specify these on the RECORD statement, which is the second argument to PLISRTx.

Finally, you must decide on the amount of main and auxiliary storage you will allow for the Sort program. For further details, see "Determining storage needed for Sort" on page 290.

Choosing the type of Sort

If you want to make the best use of the Sort program, you must understand something of how it works. In your PL/I program you specify a sort by using a CALL statement to the sort interface subroutine PLISRTx. This subroutine has four entry points: x=A, B, C, and D. Each specifies a different source for the unsorted data and destination for the data when it has been sorted. For example, a call to PLISRTA specifies that the unsorted data (the input to sort) is on a data set, and that the sorted data (the output from sort) is to be placed on another data set. The CALL PLISRTx statement must contain an argument list giving the Sort program information about the data set to be sorted, the fields on which it is to be sorted, the amount of space available, the name of a variable into which Sort will place a return code indicating the success or failure of the sort, and the name of any output or input handling procedure that can be used.

The sort interface routine builds an argument list for the Sort program from the information supplied by the PLISRTx argument list and the choice of PLISRTx entry point. Control is then transferred to the Sort program. If you have specified an output- or input-handling routine, this will be called by the Sort program as many times as is necessary to handle each of the unsorted or sorted records. When

the sort operation is complete, the Sort program returns to the PL/I calling procedure communicating its success or failure in a return code, which is placed in one of the arguments passed to the interface routine. The return code can then be tested in the PL/I routine to discover whether processing should continue. Figure 46 is a simplified flowchart showing this operation.
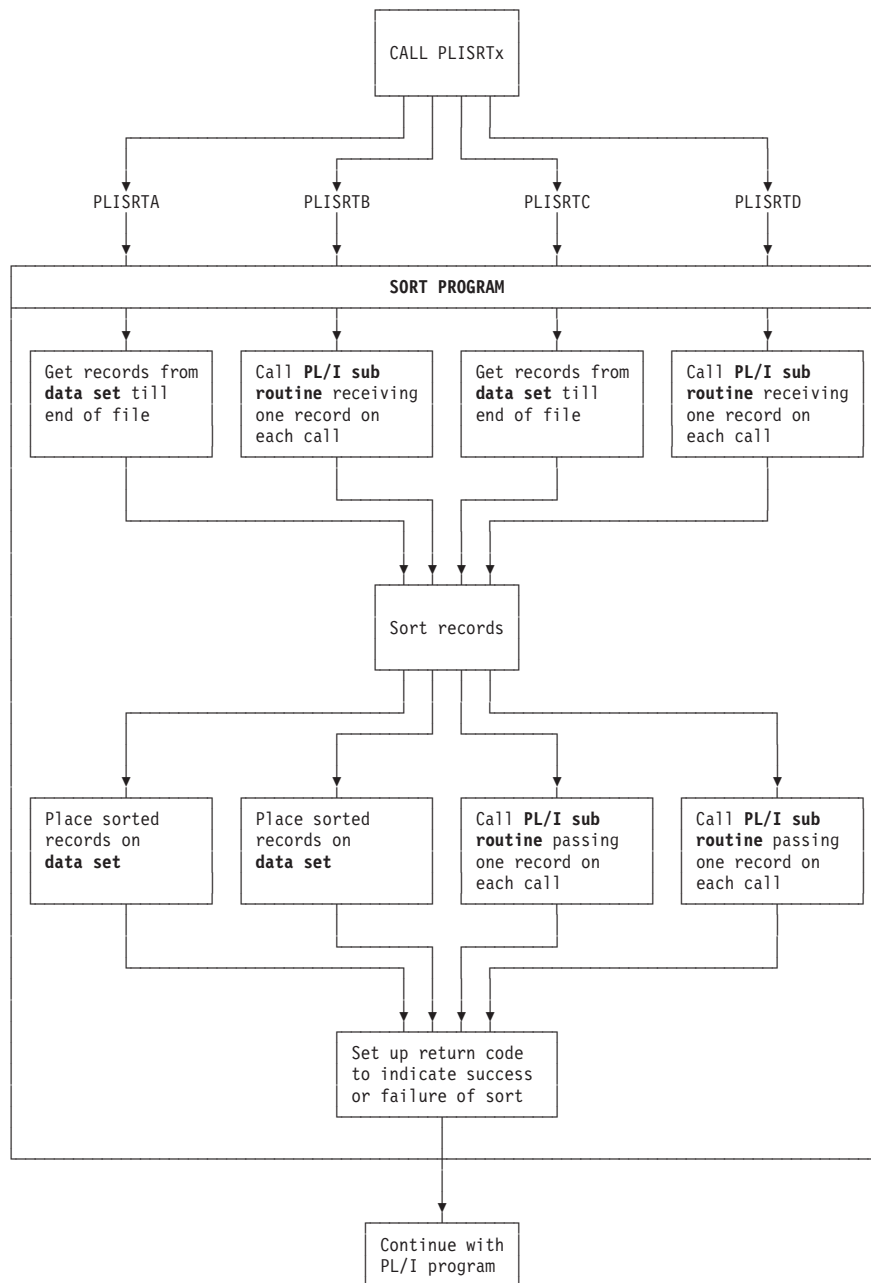
Figure 46. Flow of control for Sort program

Within the Sort program itself, the flow of control between the Sort program and input- and output-handling routines is controlled by return codes. The Sort program calls these routines at the appropriate point in its processing. (Within the Sort program, and its associated documentation, these routines are known as *user exits*. The routine that passes input to be sorted is the E15 sort user exit. The

routine that processes sorted output is the E35 sort user exit.) From the routines, Sort expects a return code indicating either that it should call the routine again, or that it should continue with the next stage of processing.

The important points to remember about Sort are: (1) it is a self-contained program that handles the complete sort operation, and (2) it communicates with the caller, and with the user exits that it calls, by means of return codes.

The remainder of this chapter gives detailed information on how to use Sort from PL/I. First the required PL/I statements are described, and then the data set requirements. The chapter finishes with a series of examples showing the use of the four entry points of the sort interface routine.

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:

```
'bSORTbFIELDS=(start1,length1,form1,seq1,
...startn,lengthn,formn,seqn)[,other options]b'
```

For example:

```
' SORT FIELDS=(1,10,CH,A) '
```

b   represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

start,length,form,seq
defines a sorting field. You can specify any number of sorting fields, but there is a limit on the total length of the fields. If more than one field is to be sorted on, the records are sorted first according to the first field, and then those that are of equal value are sorted according to the second field, and so on. If all the sorting values are equal, the order of equal records will be arbitrary unless you use the EQUALS option. (See later in this definition list.) Fields can overlay each other.

For DFSORT (5740-SM1), the maximum total length of the sorting fields is restricted to 4092 bytes and all sorting fields must be within 4092 bytes of the start of the record. Other sort products might have different restrictions.

start   is the starting position within the record. Give the value in bytes except for binary data where you can use a "byte.bit" notation. The first byte in a string is considered to be byte 1, the first bit is bit 0. (Thus the second bit in byte 2 is referred to as 2.1.) For varying length records you must include the 4-byte length prefix, making 5 the first byte of data.

length   is the length of the sorting field. Give the value in bytes except for binary where you can use "byte.bit" notation. The length of sorting fields is restricted according to their data type.

form   is the format of the data. This is the format assumed for the purpose of sorting. All data passed between PL/I routines and Sort must be in the form of character strings. The main data types and the restrictions on their length are shown below. Additional data types are available for special-purpose sorts. See the *DFSORT Application Programming Guide*, or the equivalent publication for your sort product.

Code   Data type and length

CH   character 1–4096

ZD zoned decimal, signed 1–32

PD packed decimal, signed 1–32

FI fixed point, signed 1–256

BI binary, unsigned 1 bit to 4092 bytes

FL floating-point, signed 1–256

The sum of the lengths of all fields must not exceed 4092 bytes.

seq is the sequence in which the data will be sorted as follows:

A ascending (that is, 1,2,3,...)

D descending (that is, ...,3,2,1)

> **Note:** You cannot specify E, because PL/I does not provide a method of passing a user-supplied sequence.

other options

You can specify a number of other options, depending on your Sort program. You must separate them from the FIELDS operand and from each other by commas. Do not place blanks between operands.

FILSZ=y

specifies the number of records in the sort and allows for optimization by Sort. If *y* is only approximate, E should precede *y*.

SKIPREC=y

specifies that *y* records at the start of the input file are to be ignored before sorting the remaining records.

CKPT or CHKPT

specifies that checkpoints are to be taken. If you use this option, you must provide a SORTCKPT data set and DFSORT's 16NCKPT=NO installation option must be specified.

EQUALS|NOEQUALS

specifies whether the order of equal records will be preserved as it was in the input (EQUALS) or will be arbitrary (NOEQUALS). You could improve sort performance by using the NOEQUALS. The default option is chosen when Sort is installed. The IBM-supplied default is NOEQUALS.

DYNALLOC=(d,n)

(OS/VS Sort only) specifies that the program dynamically allocates intermediate storage.
d is the device type (3380, etc.).
n is the number of work areas.

Specifying the records to be sorted

Use the RECORD statement as the second argument to PLISRTx. The syntax of the RECORD statement must be a character string expression which, when evaluated, takes the syntax shown below:

```
'bRECORDbTYPE=rectype[,LENGTH=(L1,[,,L4,L5])]b'
```

For example:

```
' RECORD TYPE=F,LENGTH=(80) '
```

b represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

TYPE

specifies the type of record as follows:

F   fixed length
V   varying length EBCDIC
D   varying length ASCII

Even when you use input and output routines to handle the unsorted and sorted data, you must specify the record type as it applies to the work data sets used by Sort.

If varying length strings are passed to Sort from an input routine (E15 exit), you should normally specify V as a record format. However, if you specify F, the records are padded to the maximum length with blanks.

LENGTH

specifies the length of the record to be sorted. You can omit LENGTH if you use PLISRTA or PLISRTC, because the length will be taken from the input data set. Note that there is a restriction on the maximum and minimum length of the record that can be sorted (see below). For varying length records, you must include the four-byte prefix.

L1   is the length of the record to be sorted. For VSAM data sets sorted as varying records it is the maximum record size+4.

,,   represent two arguments that are not applicable to Sort when called from PL/I. You must include the commas if the arguments that follow are used.

L4   specifies the minimum length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

L5   specifies the modal (most common) length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

Maximum record lengths

The length of a record can never exceed the maximum length specified by the user. The maximum record length for variable length records is 32756 bytes and for fixed length records it is 32760 bytes.

Determining storage needed for Sort

Main storage

Sort requires both main and auxiliary storage. The minimum main storage for DFSORT is 88K bytes, but for best performance, more storage (on the order of 1 megabyte or more) is recommended. DFSORT can take advantage of storage above 16M virtual or extended architecture processors. Under z/OS, DFSORT can also take advantage of expanded storage. You can specify that Sort use the maximum amount of storage available by passing a storage parameter in the following manner:

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B||UNSPEC('MAX');
CALL PLISRTA
    (' SORT FIELDS=(1,80,CH,A) ',
     ' RECORD TYPE=F,LENGTH=(80) ',
       MAXSTOR,
       RETCODE,
       'TASK');
```

If files are opened in E15 or E35 exit routines, enough residual storage should be allowed for the files to open successfully.

Auxiliary storage

Calculating the minimum auxiliary storage for a particular sorting operation is a complicated task. To achieve maximum efficiency with auxiliary storage, use direct access storage devices (DASDs) whenever possible. For more information on improving program efficiency, see the *DFSORT Application Programming Guide*, particularly the information about dynamic allocation of workspace which allows DFSORT to determine the auxiliary storage needed and allocate it for you.

If you are interested only in providing enough storage to ensure that the sort will work, make the total size of the SORTWK data sets large enough to hold three sets of the records being sorted. (You will not gain any advantage by specifying more than three if you have enough space in three data sets.)

However, because this suggestion is an approximation, it might not work, so you should check the sort manuals. If this suggestion does work, you will probably have wasted space.

Calling the Sort program

When you have determined the points described above, you are in a position to write the CALL PLISRTx statement. You should do this with some care; for the entry points and arguments to use, see Table 30.

Table 30. The entry points and arguments to PLISRTx (x = A, B, C, or D)

Entry points	Arguments
PLISRTA Sort input: data set Sort output: data set	(sort statement,record statement,storage,return code [,data set prefix,message level, sort technique])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(sort statement,record statement,storage,return code,input routine [,data set prefix,message level,sort technique])
PLISRTC Sort input: data set Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,output routine [,data set prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,input routine,output routine[,data set prefix,message level,sort technique])
Sort statement	Character string expression containing the Sort program SORT statement. Describes sorting fields and format. See "Specifying the sorting field" on page 288.
Record statement	Character string expression containing the Sort program RECORD statement. Describes the length and record format of data. See "Specifying the records to be sorted" on page 289.
Storage	Fixed binary expression giving maximum amount of main storage to be used by the Sort program. Must be >88K bytes for DFSORT. See also "Determining storage needed for Sort" on page 290.

Return code	Fixed binary variable of precision (31,0) in which Sort places a return code when it has completed. The meaning of the return code is:
	    0=Sort successful
	  16=Sort failed
	  20=Sort message data set missing

Table 30. The entry points and arguments to PLISRTx (x = A, B, C, or D) (continued)

Entry points	Arguments
Input routine	(PLISRTB and PLISRTD only.) Name of the PL/I external or internal procedure used to supply the records for the Sort program at sort exit E15.
Output routine	(PLISRTC and PLISRTD only.) Name of the PL/I external or internal procedure to which Sort passes the sorted records at sort exit E35.
Data set prefix	Character string expression of four characters that replaces the default prefix of 'SORT' in the names of the sort data sets SORTIN, SORTOUT, SORTWKnn and SORTCNTL, if used. Thus if the argument is "TASK", the data sets TASKIN, TASKOUT, TASKWKnn, and TASKCNTL can be used. This facility enables multiple invocations of Sort to be made in the same job step. The four characters must start with an alphabetic character and must not be one of the reserved names PEER, BALN, CRCX, OSCL, POLY, DIAG, SYSC, or LIST. You must code a null string for this argument if you require either of the following arguments but do not require this argument.
Message level	Character string expression of two characters indicating how Sort's diagnostic messages are to be handled, as follows: NO No messages to SYSOUT AP All messages to SYSOUT CP Critical messages to SYSOUT SYSOUT will normally be allocated to the printer, hence the use of the mnemonic letter "P". Other codes are also allowed for certain of the Sort programs. For further details on these codes, see *DFSORT Application Programming Guide*. You must code a null string for this argument if you require the following argument but you do not require this argument.
Sort technique	(This is not used by DFSORT; it appears for compatibility reasons only.) Character string of length 4 that indicates the type of sort to be carried out, as follows: PEER Peerage sort BALN Balanced CRCX Criss-cross sort OSCL Oscillating POLY Polyphase sort Normally the Sort program will analyze the amount of space available and choose the most effective technique without any action from you. You should use this argument only as a bypass for sorting problems or when you are certain that performance could be improved by another technique. See *DFSORT Application Programming Guide* for further information.

The examples below indicate the form that the CALL PLISRTx statement normally takes.

Example 1

A call to PLISRTA sorting 80-byte records from SORTIN to SORTOUT using 1048576 (1 megabyte) of storage, and a return code, RETCODE, declared as FIXED BINARY (31,0).

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE);
```

Example 2

This example is the same as example 1 except that the input, output, and work data sets are called TASKIN, TASKOUT, TASKWK01, and so forth. This might occur if Sort was being called twice in one job step.

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE,
                'TASK');
```

Example 3

This example is the same as example 1 except that the sort is to be undertaken on two fields. First, bytes 1 to 10 which are characters, and then, if these are equal, bytes 11 and 12 which contain a binary field, both fields are to be sorted in ascending order.

```
CALL PLISRTA (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE);
```

Example 4

This example shows a call to PLISRTB. The input is to be passed to Sort by the PL/I routine PUTIN, the sort is to be carried out on characters 1 to 10 of an 80 byte fixed length record. Other information as above.

```
CALL PLISRTB (' SORT FIELDS=(1,10,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                1048576,
                RETCODE,
                PUTIN);
```

Example 5

This example shows a call to PLISRTD. The input is to be supplied by the PL/I routine PUTIN and the output is to be passed to the PL/I routine PUTOUT. The record to be sorted is 84 bytes varying (including the length prefix). It is to be sorted on bytes 1 through 5 of the data in ascending order, then if these fields are equal, on bytes 6 through 10 in descending order. (Note that the 4-byte length prefix is included so that the actual values used are 5 and 10 for the starting points.) If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

```
CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(84) ',
                1048576,
                RETCODE,
                PUTIN,      /*input routine (sort exit E15)*/
                PUTOUT);    /*output routine (sort exit E35)*/
```

Determining whether the Sort was successful

When the sort is completed, Sort sets a return code in the variable named in the fourth argument of the call to 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
20 Sort message data set missing

You must declare the variable to which the return code is passed as FIXED
BINARY (31,0). It is standard practice to test the value of the return code after the
CALL 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;
```

If the job step that follows the sort depends on the success or failure of the sort,
you should set the value returned in the Sort program as the return code from the
PL/I program. This return code is then available for the following job step. The
PL/I return code is set by a call to PLIRETC. You can call PLIRETC with the value
returned from Sort thus:

```
CALL PLIRETC(RETCODE);
```

You should not confuse this call to PLIRETC with the calls made in the input and
output routines, where a return code is used for passing control information to
Sort.

Establishing data sets for Sort

If DFSORT was installed in a library not known to the system, you must specify
the DFSORT library in a JOBLIB or STEPLIB DD statement.

When you call Sort, certain sort data sets must not be open. These are:

SYSOUT
A data set (normally the printer) on which messages from the Sort program
will be written.

Sort work data sets

SORTWK01–SORTWK32

> **Note:** If you specify more than 32 sort work data sets, DFSORT will only use
> the first 32.

****WK01–****WK32
From 1 to 32 working data sets used in the sorting process. These must be
direct access. For a discussion of space required and number of data sets, see
"Determining storage needed for Sort" on page 290.

**** represents the four characters that you can specify as the data set prefix
argument in calls to PLISRTx. This allows you to use data sets with prefixes
other than SORT. They must start with an alphabetic character and must not be
the names PEER, BALN, CRCX, OSCL, POLY, SYSC, LIST, or DIAG.

Input data set

SORTIN

****IN
The input data set used when PLISRTA and PLISRTC are called.

See *****WK01–****WK32* above for a detailed description.

Output data set

SORTOUT

****OUT
> The output data set used when PLISRTA and PLISRTB are called.
>
> See *****WK01–****WK32* above for a detailed description.

Checkpoint data set

SORTCKPT

> Data set used to hold checkpoint data, if CKPT or CHKPT option was used in the SORT statement argument and DFSORT's 16NCKPT=NO installation option was specified. For information on this program DD statement, see *DFSORT Application Programming Guide.*

DFSPARM SORTCNTL

> Data set from which additional or changed control statements can be read (optional). For additional information on this program DD statement, see *DFSORT Application Programming Guide.*
>
> See *****WK01–****WK32* above for a detailed description.

Sort data input and output

The source of the data to be sorted is provided either directly from a data set or indirectly by a routine (Sort Exit E15) written by the user. Similarly, the destination of the sorted output is either a data set or a routine (Sort Exit E35) provided by the user.

PLISRTA is the simplest of all of the interfaces because it sorts from data set to data set. An example of a PLISRTA program is in Figure 50 on page 300. Other interfaces require either the input handling routine or the output handling routine, or both.

Data input and output handling routines

The input handling and output handling routines are called by Sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRTx, they behave in the same way as ordinary internal procedures in respect of scope of names. The input and output procedure names must themselves be known in the procedure that makes the call to PLISRTx.

The routines are called individually for each record required by Sort or passed from Sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures will not retain their values between calls. Consequently, items such as counters, which need to be retained from one call to the next, should either be declared as STATIC or be declared in the containing block.

The E15 and E35 sort exits must not be MAIN procedures.
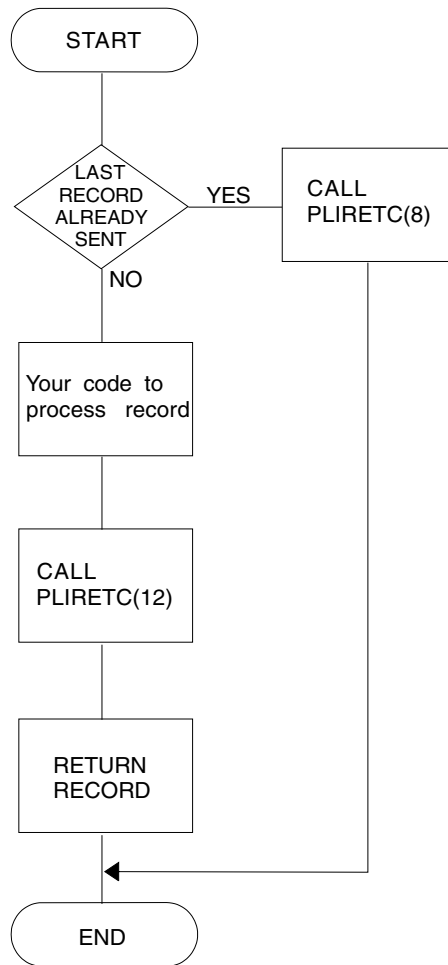
E15—Input handling routine (Sort Exit E15)

Input routines are normally used to process the data in some way before it is sorted. You can use input routines to print the data, as shown in the Figure 51 on page 301 and Figure 53 on page 303, or to generate or manipulate the sorting fields to achieve the correct results.

The input handling routine is used by Sort when a call is made to either PLISRTB or PLISRTD. When Sort requires a record, it calls the input routine which should return a record in character string format, with a return code of 12. This return code means that the record passed is to be included in the sort. Sort continues to call the routine until a return code of 8 is passed. A return code of 8 means that all records have already been passed, and that Sort is not to call the routine again. If a record is returned when the return code is 8, it is ignored by Sort.

The data returned by the routine must be a character string. It can be fixed or varying. If it is varying, you should normally specify V as the record format in the RECORD statement which is the second argument in the call to PLISRTx. However, you can specify F, in which case the string will be padded to its maximum length with blanks. The record is returned with a RETURN statement, and you must specify the RETURNS attribute in the PROCEDURE statement. The return code is set in a call to PLIRETC. A flowchart for a typical input routine is shown in Figure 47 on page 297.

Input Handling Subroutine

Output Handling Subroutine

```
          ┌───────────┐
          │   START   │
          └───────────┘
                │
             ╱  LAST  ╲
            ╱  RECORD  ╲      YES     ┌──────────────┐
           ╱  ALREADY   ╲─────────────│ CALL         │
            ╲   SENT    ╱             │ PLIRETC(8)   │
             ╲        ╱               └──────────────┘
                │                            │
               NO                           │
                │                           │
        ┌───────────────┐                   │
        │ Your code to  │                   │
        │ process record│                   │
        └───────────────┘                   │
                │                           │
        ┌───────────────┐                   │
        │ CALL          │                   │
        │ PLIRETC(12)   │                   │
        └───────────────┘                   │
                │                           │
        ┌───────────────┐                   │
        │ RETURN        │                   │
        │ RECORD        │                   │
        └───────────────┘                   │
                │◄──────────────────────────┘
          ┌───────────┐
          │    END    │
          └───────────┘
```

```
          ┌───────────┐
          │   START   │
          └───────────┘
                │
        ┌───────────────┐
        │ RECEIVE       │
        │ RECORD        │
        │ PARAMETER     │
        └───────────────┘
                │
        ┌───────────────┐
        │ Your code to  │
        │ process record│
        └───────────────┘
                │
        ┌───────────────┐
        │ CALL          │
        │ PLIRETC(4)    │
        └───────────────┘
                │
          ┌───────────┐
          │    END    │
          └───────────┘
```
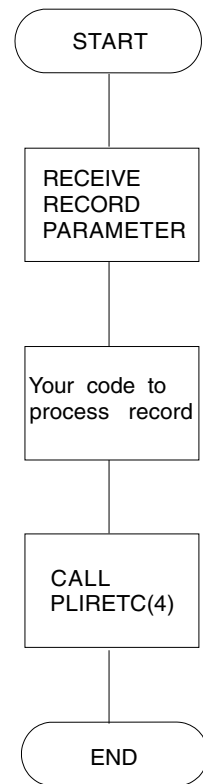
Figure 47. Flowcharts for input and output handling subroutines

Skeletal code for a typical input routine is shown in Figure 48 on page 298.

```
E15: PROC RETURNS (CHAR(80));
    /*----------------------------------------------------------------*/
    /*RETURNS attribute must be used specifying length of data to be */
    /* sorted, maximum length if varying strings are passed to Sort. */
    /*----------------------------------------------------------------*/
  DCL STRING CHAR(80); /*---------------------------------------------*/
                       /*A character string variable will normally be*/
                       /* required to return the data to Sort        */
                       /*--------------------------------------------*/

  IF LAST_RECORD_SENT THEN
    DO;
    /*--------------------------------------------------------------*/
    /*A test must be made to see if all the records have been sent, */
    /*if they have, a return code of 8 is set up and control returned*/
    /*to Sort                                                       */
    /*--------------------------------------------------------------*/

       CALL PLIRETC(8);  /*-------------------------------------------*/
                         /* Set return code of 8, meaning last record */
                         /* already sent.                            */
                         /*------------------------------------------*/
       GOTO FINAL;

    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;
FINAL:
  END;   /*End of the input procedure*/
```

Figure 48. Skeletal code for an input procedure

Examples of an input routine are given in Figure 51 on page 301 and Figure 53 on page 303.

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 causes Sort to return to your PL/I program with a return code of 16–Sort failed.

Note: A call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When an output handling routine has been used, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRTx to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 52 on page 302.

E35—Output handling routine (Sort Exit E35)

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 52 on page 302 and Figure 53 on page 303, 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 PLISRTx statement. There is no indication from Sort to the output handling routine that the last record has been reached. Any end-of-data handling must therefore be done in the procedure that calls PLISRTx.

The record is passed from Sort to the output routine as a character string, and you must declare a character string parameter in the output handling subroutine to receive the data. The output handling subroutine must also pass a return code of 4 to Sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

The sort can be stopped by passing a return code of 16 to Sort. This will result in Sort returning to the calling program with a return code of 16–Sort failed.

The record passed to the routine by Sort is a character string parameter. If you specified the record type as F in the second argument in the call to PLISRTx, you should declare the parameter with the length of the record. If you specified the record type as V, you should declare the parameter as adjustable, as in the following example:

```
DCL STRING CHAR(*);
```

Figure 54 on page 305 shows a program that sorts varying length records.

A flowchart for a typical output handling routine is given in Figure 47 on page 297. Skeletal code for a typical output handling routine is shown in Figure 49.

```
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 49. Skeletal code for an output handling procedure

You should note that a call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When you have used an output handling routine, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRTx to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples at the end of this chapter.

Calling PLISRTA example

After each time that the PL/I input- and output-handling routines communicate the return-code information to the Sort program, the return-code field is reset to zero; therefore, it is not used as a regular return code other than its specific use for the Sort program.

For details on handling conditions, especially those that occur during the input- and output-handling routines, see *z/OS Language Environment Programming Guide.*

```
//OPT14#7  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX106: PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                     1048576
                     RETURN_CODE);
     SELECT (RETURN_CODE);
       WHEN(0) PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT (
            'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
       CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/
       END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*
```

Figure 50. PLISRTA—sorting from input data set to output data set

Calling PLISRTB example

```
//OPT14#8  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX107:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X);
     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT
           ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
     END /* select */;
     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

  E15X:   /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT
              STREAM AND PUTS THEM BEFORE THEY ARE SORTED*/
     PROC RETURNS (CHAR(80));
         DCL SYSIN FILE RECORD INPUT,
             INFIELD CHAR(80);

         ON ENDFILE(SYSIN) BEGIN;
            PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
            CALL PLIRETC(8);  /* signal that last record has
                                 already been sent to sort*/
            GOTO ENDE15;
            END;

          READ FILE (SYSIN) INTO (INFIELD);
          PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
          CALL PLIRETC(12);  /* request sort to include current
                                record and return for more*/
         RETURN(INFIELD);
    ENDE15:
         END E15X;
  END EX107;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT  DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 51. PLISRTB—sorting from input handling routine to output data set

Calling PLISRTC example

```
//OPT14#9  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX108:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E35X);
       SELECT(RETURN_CODE);
         WHEN(0)  PUT SKIP EDIT
             ('SORT COMPLETE RETURN_CODE 0') (A);
         WHEN(16) PUT SKIP EDIT
             ('SORT FAILED, RETURN_CODE 16') (A);
         WHEN(20) PUT SKIP EDIT
             ('SORT MESSAGE DATASET MISSING ') (A);
         OTHER    PUT SKIP EDIT
             ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
     CALL PLIRETC (RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

  E35X:   /* output handling routine prints sorted records*/
     PROC (INREC);
        DCL INREC CHAR(80);
        PUT SKIP EDIT (INREC) (A);
        CALL PLIRETC(4); /*request next record from sort*/
     END E35X;
   END EX108;
/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT    DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 52. PLISRTC—sorting from input data set to output handling routine

Calling PLISRTD example

```
//OPT14#10  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX109:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);
     CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X,
                   E35X);

     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT
           ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;

     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

  E15X:   /* Input handling routine prints input before sorting*/
      PROC RETURNS(CHAR(80));
          DCL INFIELD CHAR(80);

          ON ENDFILE(SYSIN) BEGIN;
             PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT.  ',
                 'SORTED OUTPUT SHOULD FOLLOW')(A);
             CALL PLIRETC(8);  /* Signal end of input to sort*/
             GOTO ENDE15;
          END;

          GET FILE (SYSIN) EDIT (INFIELD) (A(80));
          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 53. PLISRTD—sorting from input handling routine to output handling routine (Part 1 of 2)

```
 /*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
```

Figure 53. PLISRTD—sorting from input handling routine to output handling routine (Part 2 of 2)

Sorting variable-length records example

```
//OPT14#11 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
  /* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH
     RECORDS */

  EX1306:  PROC OPTIONS(MAIN);
        DCL RETURN_CODE FIXED BIN(31,0);
        CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
                      ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                      /*NOTE THAT LENGTH IS MAX AND INCLUDES
                        4 BYTE LENGTH PREFIX*/
                      1048576
                      RETURN_CODE,
                      PUTIN,
                      PUTOUT);

            SELECT(RETURN_CODE);
              WHEN(0)  PUT SKIP EDIT (
                    'SORT COMPLETE RETURN_CODE 0') (A);
              WHEN(16) PUT SKIP EDIT (
                    'SORT FAILED, RETURN_CODE 16') (A);
              WHEN(20) PUT SKIP EDIT (
                    'SORT MESSAGE DATASET MISSING ') (A);
              OTHER    PUT SKIP EDIT (
                  'INVALID RETURN_CODE = ', RETURN_CODE)
                      (A,F(2));
            END /* SELECT */;

     CALL PLIRETC(RETURN_CODE);
      /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
      PUTIN: PROC RETURNS (CHAR(80) VARYING);
        /*OUTPUT HANDLING ROUTINE*/
        /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE
          WHEN USING VARYING LENGTH RECORDS*/
          DCL STRING CHAR(80) VAR;

          ON ENDFILE(SYSIN) BEGIN;
              PUT SKIP EDIT ('END OF INPUT')(A);
              CALL PLIRETC(8);
              GOTO ENDPUT;
              END;

          GET EDIT(STRING)(A(80));
          I=INDEX(STRING||' ',' ')-1;/*RESET LENGTH OF THE*/
          STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO */
                                       /* LENGTH OF TEXT IN */
                                       /* EACH INPUT RECORD.*/
```

Figure 54. Sorting varying-length records using input and output handling routines (Part 1 of 2)

```
              PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
              CALL PLIRETC(12);
              RETURN(STRING);
 ENDPUT:  END;
 PUTOUT:PROC(STRING);
           /*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
           DCL STRING CHAR (*);
            /*NOTE THAT FOR VARYING RECORDS THE STRING
               PARAMETER FOR THE OUTPUT-HANDLING ROUTINE
               SHOULD BE DECLARED ADJUSTABLE BUT CANNOT BE
               DECLARED VARYING*/
           PUT SKIP EDIT(STRING)(A);  /*PRINT THE SORTED DATA*/
           CALL PLIRETC(4);
           END;  /*ENDS PUTOUT*/
          END;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//*
```

Figure 54. Sorting varying-length records using input and output handling routines (Part 2 of 2)

Chapter 13. ILC with C

This chapter describes some aspects of InterLanguage Communication (ILC) between PL/I and C. The examples illustrate how to use many of the data types common to both languages and should enable you to write PL/I code that either calls or is called by C.

Equivalent data types

The table Table 31 lists the common C and PL/I data type equivalents.

Table 31. C and PL/I Type Equivalents

C type	Matching PL/I type
char[...]	char(...) varyingz
wchar[...]	wchar(...) varyingz
signed char	fixed bin(7)
unsigned char	unsigned fixed bin(8)
short	fixed bin(15)
unsigned short	unsigned fixed bin(16)
int	fixed bin(31)
unsigned int	unsigned fixed bin(32)
long long	fixed bin(63)
unsigned long long	unsigned fixed bin(64)
float	float bin(21)
double	float bin(53)
long double	float bin(p) (p >= 54)
enum	ordinal
typedef	define alias
struct	define struct
union	define union
struct *	handle

Simple type equivalence

So, for example, the following illustrates the translation of the simple typedef for *time_t* from the C header file *time.h*:

```
typedef long time_t;

define alias time_t fixed bin(31);
```

Figure 55. Simple type equivalence

Struct type equivalence

The following example illustrates the translation of the simple struct for *tm* from the C header file *time.h*:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};

define structure
  1 tm
    ,2 tm_sec   fixed bin(31)
    ,2 tm_min   fixed bin(31)
    ,2 tm_hour  fixed bin(31)
    ,2 tm_mday  fixed bin(31)
    ,2 tm_mon   fixed bin(31)
    ,2 tm_year  fixed bin(31)
    ,2 tm_wday  fixed bin(31)
    ,2 tm_yday  fixed bin(31)
    ,2 tm_isdst fixed bin(31)
   ;
```

Figure 56. Sample struct type equivalence

Enum type equivalence

The following example illustrates the translation of the simple enum *__device_t* from the C header file *stdio.h*:

```
typedef enum {
    __disk       =  0,
    __terminal   =  1,
    __printer    =  2,
    __tape       =  3,
    __tdq        =  5,
    __dummy      =  6,
    __memory     =  8,
    __hfs        =  9,
    __hiperspace = 10
} __device_t;

define ordinal __device_t (
       __disk      value(0)
    , __terminal   value(1)
    , __printer    value(2)
    , __tape       value(3)
    , __tdq        value(4)
    , __dummy      value(5)
    , __memory     value(8)
    , __hfs        value(9)
    , __hiperspace value(10)
);
```

Figure 57. Sample enum type equivalence

File type equivalence

A C file declaration depends on the platform, but it often starts as follows:

```
struct __file {
    unsigned char  *__bufPtr;
 ... } FILE;
```

Figure 58. Start of the C declaration for its FILE type

All we want is a pointer (or token) for a file, so we can finesse this translation with:

```
define struct    1 file;
define alias      file_Handle  handle file;
```

Figure 59. PL/I equivalent for a C file

Using C functions

Let's say we wanted to write a program to read a file and dump it as formatted hex - using the C functions fopen and fread.

The code for this program is straightforward:

```
filedump:
  proc(fn) options(noexecops main);

  dcl fn            char(*) var;

  %include filedump;

  file = fopen( fn, 'rb' );

  if file = sysnull() then
    do;
      display( 'file could not be opened' );
      return;
    end;

  do forever;
    unspec(buffer) = ''b;

    read_In = fread( addr(buffer), 1, stg(buffer), file );

    if read_In = 0 then
      leave;

    display(   heximage(addr(buffer),16,' ')  ||  ' '
            || translate(buffer,(32)'.',unprintable)  );

    if read_In < stg(buffer) then
      leave;
  end;

  call fclose( file );
end filedump;
```

Figure 60. Sample code to use fopen and fread to dump a file

Most of the declarations in the INCLUDE file *filedump* are obvious:

```
define struct       1 file;
define alias        file_Handle  handle file;

define alias        size_t unsigned fixed bin(32);
define alias        int signed fixed bin(31);

dcl file            type(file_Handle);
dcl read_In         fixed bin(31);
dcl buffer          char(16);

dcl unprintable     char(32) value( substr(collate(),1,32) );
```

Figure 61. Declarations for filedump program

Matching simple parameter types

It would be easy to mistranslate the declarations for the C functions. For instance, you could take the following declaration for the C function *fread*:

```
size_t    fread(  void *,
                  size_t,
                  size_t,
                  FILE *);
```

Figure 62. C declaration of fread

and translate it to:

```
dcl fread     ext
              entry( pointer,
                     type size_t,
                     type size_t,
                     type file_Handle )
              returns( type size_t );
```

Figure 63. First incorrect declaration of fread

On some platforms, this would not link successfully because C names are case
sensitive. In order to prevent this kind of linker problem, it is best to specify the
name in mixed case using the extended form of the *external* attribute. So, for
instance, the declare for *fread* would be better as:

```
dcl fread     ext('fread')
              entry( pointer,
                     type size_t,
                     type size_t,
                     type file_Handle )
              returns( type size_t );
```

Figure 64. Second incorrect declaration of fread

But this wouldn't run right, because while PL/I parameters are *byaddr* by default,
C parameters are *byvalue* by default; so we fix that by adding the *byvalue* attribute
to the parameters:

```
dcl fread     ext('fread')
              entry( pointer byvalue,
                     type size_t byvalue,
                     type size_t byvalue,
                     type file_Handle byvalue )
              returns( type size_t );
```

Figure 65. Third incorrect declaration of fread

But note how the return value is set in Figure 66 on page 312: a fourth parameter
(the address of the temporary *_temp5*) is passed to the function *fread*, which is then
expected to place the return code in the integer at that address. This is the
convention for how values are returned when the *byaddr* attribute applies to
returns, and PL/I uses this convention by default.

```
*        read_In = fread( addr(buffer), 1, stg(buffer), file );
*
         L     r4,FILE(,r13,176)
         L     r1,fread(,r5,12)
         LA    r2,_temp5(,r13,420)
         LA    r8,BUFFER(,r13,184)
         L     r15,&EPA_&WSA(,r1,8)
         L     r0,&EPA_&WSA(,r1,12)
         ST    r0,_CEECAA_(,r12,500)
         LA    r1,#MX_TEMP1(,r13,152)
         ST    r8,#MX_TEMP1(,r13,152)
         LA    r8,1
         ST    r8,#MX_TEMP1(,r13,156)
         ST    r7,#MX_TEMP1(,r13,160)
         ST    r4,#MX_TEMP1(,r13,164)
         ST    r2,#MX_TEMP1(,r13,168)
         BALR  r14,r15
         L     r0,_temp5(,r13,420)
         ST    r0,READ_IN(,r13,180)
```

Figure 66. Code generated for RETURNS BYADDR

This wouldn't run right, because C return values are *byvalue*. So, we fix that with one more *byvalue* attribute.

```
  dcl fread    ext('fread')
               entry( pointer byvalue,
                      type size_t byvalue,
                      type size_t byvalue,
                      type file_Handle byvalue )
               returns( type size_t byvalue );
```

Figure 67. Correct declaration of fread

Note how the return value is set now in Figure 68: no extra address is passed, and the return value is simply returned in register 15.

```
*        read_In = fread( addr(buffer), 1, stg(buffer), file );
*
         L     r2,FILE(,r13,176)
         L     r1,fread(,r5,12)
         LA    r7,BUFFER(,r13,184)
         L     r15,&EPA_&WSA(,r1,8)
         L     r0,&EPA_&WSA(,r1,12)
         ST    r0,_CEECAA_(,r12,500)
         LA    r1,#MX_TEMP1(,r13,152)
         ST    r7,#MX_TEMP1(,r13,152)
         LA    r7,1
         ST    r7,#MX_TEMP1(,r13,156)
         ST    r4,#MX_TEMP1(,r13,160)
         ST    r2,#MX_TEMP1(,r13,164)
         BALR  r14,r15
         LR    r0,r15
         ST    r0,READ_IN(,r13,180)
```

Figure 68. Code generated for RETURNS BYVALUE

Matching string parameter types

Now that we have translated *fread* correctly, we might try this translation for *fopen*:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byvalue,
                    char(*) varyingz byvalue )
             returns( byvalue type file_handle );
```

Figure 69. First incorrect declaration of fopen

But C really has no strings, only pointers, and these pointers would be passed *byvalue*; so the strings should be *byaddr*:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
             returns( byvalue type file_handle );
```

Figure 70. Second incorrect declaration of fopen

But PL/I passes descriptors with strings and C doesn't understand them, so they must be suppressed. We can do this by adding *options(nodescriptor)* to the declaration:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
             returns( byvalue type file_handle )
             options ( nodescriptor );
```

Figure 71. Correct declaration of fopen

This will work, but isn't optimal since the parameters are input-only; if the parameter is a constant, the *nonassignable* attribute will prevent a copy being made and passed. Hence, the best translation of the declaration of *fopen* is:

```
dcl fopen    ext('fopen')
             entry( char(*) varyingz nonasgn byaddr,
                    char(*) varyingz nonasgn byaddr )
             returns( byvalue type file_handle )
             options ( nodescriptor );
```

Figure 72. Optimal, correct declaration of fopen

At this point, the declare for the *fclose* function presents few surprises except perhaps for the *optional* attribute in the returns specification. This attribute allows us to invoke the fclose function via a CALL statement and not have to dispose of the return code. But please note that if the file were an output file, the return code on fclose should always be checked since the last buffer may be written out only when the file is closed and that write could fail for lack of space.

```
dcl fclose   ext('fclose')
             entry( type file_handle byvalue )
             returns( optional type int byvalue )
             options ( nodescriptor );
```

Figure 73. Declaration of fclose

Now, on z/OS UNIX, we can compile and run the programs with the commands:

```
pli -qdisplay=std filedump.pli

filedump filedump.pli
```

Figure 74. Commands to compile and run filedump

This would produce the following output:

```
15408689 938584A4 94977A40 97999683  . filedump: proc
4D86955D 409697A3 899695A2 4D959685  (fn) options(noe
A7858396 97A24094 8189955D 5E151540  xecops main);..
```

Figure 75. Output of running filedump

Functions returning ENTRYs

The C quicksort function *qsort* takes a compare routine. For instance, to sort an array of integers, the following function (which use the *byvalue* attribute twice - for the reasons discussed above) could be used:

```
comp2:
  proc( key, element )
  returns( fixed bin(31) byvalue );

  dcl (key, element) pointer byvalue;
  dcl word based fixed bin(31);

  select;
    when( key->word < element->word )
      return( -1 );
    when( key->word = element->word )
      return( 0 );
    otherwise
      return( +1 );
  end;
end;
```

Figure 76. Sample compare routine for C qsort function

And the C *qsort* function could be used with this compare routine to sort an array of integers, as in the following code fragment:

```
dcl a(1:4) fixed bin(31) init(19,17,13,11);

put skip data( a );

call qsort( addr(a), dim(a), stg(a(1)), comp2 );

put skip data( a );
```

Figure 77. Sample code to use C qsort function

But since C function pointers are not the same as PL/I ENTRY variables, the C *qsort* function must not be declared as simply:

```
dcl qsort    ext('qsort')
             entry( pointer,
                    fixed bin(31),
                    fixed bin(31),
                    entry returns( byvalue fixed bin(31) )
                  )
             options( byvalue nodescriptor );
```

Figure 78. Incorrect declaration of qsort

Recall that a PL/I ENTRY variable may point to a nested function (and thus requires a backchain address as well as an entry point address). But a C function pointer is limited in pointing to a non-nested function only and so, a PL/I ENTRY variable and a C function pointer do not even use the amount of storage.

However, a C function pointer is equivalent to the new PL/I type: a LIMITED ENTRY. and hence the C *qsort* function could be declared as:

```
dcl qsort    ext('qsort')
             entry( pointer,
                    fixed bin(31),
                    fixed bin(31),
                    limited entry
                    returns( byvalue fixed bin(31) )
                  )
             options( byvalue nodescriptor );
```

Figure 79. Correct declaration of qsort

Linkages

On z/OS, there are two crucial facts about linkages:
- IBM C, JAVA and Enterprise PL/I use the same linkage by default.
- This linkage is not the system linkage.

For a traditional PL/I application where all parameters are *byaddr*, the differences between the code generated when a function has the default linkage and when it has the system linkage would usually not matter. But if the parameters are *byvalue* (as they usually are in C and JAVA), the differences can break your code.

In fact, there is only a small difference if the parameters are *byaddr*. In Figure 80, the only difference between the code generated for a function with the default linkage and for one with the system linkage is that the high-order bit is turned on for the last parameter of the system linkage call.

This difference would be transparent to most programs.

```
dcl dftv  ext entry( fixed bin(31) byaddr
                    ,fixed bin(31) byaddr );
dcl sysv  ext entry( fixed bin(31) byaddr
                    ,fixed bin(31) byaddr )
          options( linkage(system) );

*       call dfta( n, m );
*
        LA    r0,M(,r13,172)
        LA    r2,N(,r13,168)
        L     r15,=V(DFTV)(,r3,126)
        LA    r1,#MX_TEMP1(,r13,152)
        ST    r2,#MX_TEMP1(,r13,152)
        ST    r0,#MX_TEMP1(,r13,156)
        BALR  r14,r15
*
*       call sysa( n, m );
*
        LA    r0,M(,r13,172)
        LA    r2,N(,r13,168)
        O     r0,=X'80000000'
        L     r15,=V(SYSV)(,r3,130)
        LA    r1,#MX_TEMP1(,r13,152)
        ST    r2,#MX_TEMP1(,r13,152)
        ST    r0,#MX_TEMP1(,r13,156)
        BALR  r14,r15
```

Figure 80. Code when parameters are BYADDR

But, there is a big difference if the parameters are *byvalue* rather than *byaddr*. In Figure 81 on page 317, for the function with the default linkage, register 1 points to the values of the integers passed, while for the function with the system linkage, register 1 points to the addresses of those values.

This difference would *not* be transparent to most programs.

```
        dcl dftv  ext entry( fixed bin(31) byvalue
                           ,fixed bin(31) byvalue );
        dcl sysv  ext entry( fixed bin(31) byvalue
                           ,fixed bin(31) byvalue )
                  options( linkage(system) );

        *      call dftv( n, m );
        *
                L      r2,N(,r13,168)
                L      r0,M(,r13,172)
                L      r15,=V(DFTV)(,r3,174)
                LA     r1,#MX_TEMP1(,r13,152)
                ST     r2,#MX_TEMP1(,r13,152)
                ST     r0,#MX_TEMP1(,r13,156)
                BALR   r14,r15
        *
        *      call sysv( n, m );
        *
                L      r1,N(,r13,168)
                L      r0,M(,r13,172)
                ST     r0,#wtemp_1(,r13,176)
                LA     r0,#wtemp_1(,r13,176)
                ST     r1,#wtemp_2(,r13,180)
                LA     r2,#wtemp_2(,r13,180)
                O      r0,=X'80000000'
                L      r15,=V(SYSV)(,r3,178)
                LA     r1,#MX_TEMP1(,r13,152)
                ST     r2,#MX_TEMP1(,r13,152)
                ST     r0,#MX_TEMP1(,r13,156)
                BALR   r14,r15
```

Figure 81. Code when parameters are BYVALUE

Sharing output

If you want to share SYSPRINT with a C program, you must compile your PL/I code with the STDSYS option.

By default, DISPLAY statements use WTO's to display their output. If you specify the DISPLAY(STD) compiler option, DISPLAY statements will use the C puts function to display their output. This can be particularly useful under z/OS UNIX.

Summary

What we have learned in this chapter:
- C is case sensitive.
- Parameters should be BYVALUE.
- Return values should be BYVALUE.
- String parameters should be BYADDR.
- Arrays and structures should also be BYADDR.
- No descriptors should be passed.
- Input-only strings should be NONASSIGNABLE.
- C function pointers map to LIMITED ENTRYs.
- The IBM C compilers and the IBM PL/I compilers use the same default linkage (and it matters).

Chapter 14. Interfacing with Java

This chapter gives a brief description of Java and the Java Native Interface (JNI) and explains why you might be interested in using it with PL/I. A simple Java - PL/I application will be described and information on compatibility between the two languages will also be discussed. Instructions on how to build and run the Java - PL/I sample applications assume the work is being done in the z/OS UNIX System Services environment of z/OS.

Before you can communicate with Java from PL/I you need to have Java installed on your z/OS system. Contact your local System Administrator for more information on how to set up your z/OS Java environment.

These sample programs have been compiled and tested with Java 2 Version 1.4.2. To determine the level of Java in your z/OS UNIX System Services environment enter this command from the command line:

```
java -version
```

The active Java version will then be displayed and will look something like:

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM z/OS Persistent Reusable VM build cm142
```

What is the Java Native Interface (JNI)?

Java is an object-oriented programming language invented by Sun Microsystems and provides a powerful way to make Internet documents interactive.

The Java Native Interface (JNI) is the Java interface to native programming languages and is part of the Java Development Kits. By writing programs that use the JNI, you ensure that your code is portable across many platforms.

The JNI allows Java code that runs within a Java Virtual Machine (JVM) to operate with applications and libraries written in other languages, such as PL/I. In addition, the *Invocation API* allows you to embed a Java Virtual Machine into your native PL/I applications.

Java is a fairly complete programming language; however, there are situations in which you want to call a program written in another programming language. You would do this from Java with a method call to a native language, known as a *native method*.

Some reasons to use native methods may include the following:
- The native language has a special capability that your application needs and that the standard Java class libraries lack.
- You already have many existing applications in your native language and you wish to make them accessible to a Java application.
- You wish to implement a intensive series of complicated calculations in your native language and have your Java applications call these functions.
- You or your programmers have a broader skill set in your native language and you do not wish to loose this advantage.

Programming through the JNI lets you use native methods to do many different operations. A native method can:

- utilize Java objects in the same way that a Java method uses these objects.
- create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks.
- inspect and use objects created by Java application code.
- update Java objects that it created or were passed to it, and these updated objects can then be made available to the Java application.

Finally, native methods can also easily call already existing Java methods, capitalizing on the functionality already incorporated in the Java programming framework. In these ways, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

JNI Sample Program #1 - 'Hello World'

Writing Java Sample Program #1

The first sample program is yet another variation of the ″Hello World!″ program.

The ″Hello World!″ program has one Java class, *callingPLI.java*. The native method, written in PL/I, is contained in *hiFromPLI.pli*. Here is a brief overview of the steps for creating this sample program:

1. Write a Java program that defines a class containing a native method, loads the native load library, and calls the native method.
2. Compile the Java program to create a Java class.
3. Write a PL/I program that implements the native method and displays the ″Hello!″ text.
4. Compile and link the PL/I program.
5. Run the Java program which calls the native method in the PL/I program.

Step 1: Writing the Java Program

Declare the Native Method: All methods, whether Java methods or native methods, must be declared within a Java class. The only difference in the declaration of a Java method and a native method is the keyword *native*. The *native* keyword tells Java that the implementation of this method will be found in a native library that will be loaded during the execution of the program. The declaration of the native method looks like this:

```
public native void callToPLI();
```

In the above statement, the *void* means that there is no return value expected from this native method call. The empty parentheses in the method name *callToPLI( )*, means that there are no parameters being passed on the call to the native method.

Load the Native Library: A step that loads the native library must be included so the native library will be loaded at execution time. The Java statement that loads the native library looks like this:

```
 static {
   System.loadLibrary("hiFromPLI");
 }
```

In the above statement, the Java System method *System.loadLibrary(...)* is called to find and load the native library. The PL/I shared library, *libhiFromPLI.so*, will created during the step that compiles and links the PL/I program.

Write the Java Main Method: The *callingPLI* class also includes a *main* method to instantiate the class and call the native method. The *main* method instantiates *callingPLI* and calls the *callToPLI()* native method.

The complete definition of the *callingPLI* class, including all the points addressed above in this section, looks like this:

```
public class callingPLI {
   public native void callToPLI();
   static {
      System.loadLibrary("hiFromPLI");
   }
   public static void main(String[] argv) {
      callingPLI callPLI = new callingPLI();
      callPLI.callToPLI();
      System.out.println("And Hello from Java, too!");
   }
}
```

Step 2: Compiling the Java Program

Use the Java compiler to compile the *callingPLI* class into an executable form. The command would look like this:

```
javac callingPLI.java
```

Step 3: Writing the PL/I Program

The PL/I implementation of the native method looks much like any other PL/I subroutine.

Useful PL/I Compiler Options: The sample program contains a series of **PROCESS* statements that define the important compiler options.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Default( ASCII IEEE );
```

Here is a brief description of them and why they are useful:

Extname(100)
>   Allows for longer, Java style, external names.

Margins(1,100)
>   Extending the margins gives you more room for Java style names and identifiers.

Display(Std)
>   Writes the ″Hello World″ text to stdout, not via WTOs. In the z/OS UNIX environment WTOs would not be seen by the user.

Dllinit
>   Includes the initialization coded needed for creating a DLL.

Extrn(Short)
>   EXTRNs are emitted only for those constants that are referenced. This option is necessary for Enterprise PL/I V3R3 and later.

Default(ASCII IEEE);
>   *ASCII* specifies that CHARACTER and PICTURE data is held in ASCII - the form in which it is held by JAVA

IEEE specifies that FLOAT data is held in IEEE format - the form in which it is held by JAVA

The following default PL/I compiler option is also necessary. A Java - PL/I program may not work correctly if this option is changed or overridden.

RENT
    The RENT option insures that code is reentrant even if it writes on static variables.

Correct Form of PL/I Procedure Name and Procedure Statement: The PL/I procedure name must conform to the Java naming convention in order to be located by the Java Class Loader at execution time. The Java naming scheme consists of three parts. The first part identifies the routine to the Java environment, the second part is the name of the Java class that defines the native method, and the third part is the name of the native method itself.

Here is a breakdown of PL/I procedure name *Java_callingPLI_callToPLI* in the sample program:

Java
    All native methods resident in dynamic libraries must begin with *Java*

_callingPLI
    The name of the Java class that declares the native method

_callToPLI
    The name of the native method itself.

Note: There is an important difference between coding a native method in PL/I and in C. The *javah* tool, which is shipped with the JDK, generates the form of the external references required for C programs. When you write your native methods in PL/I and follow the rules above for naming your PL/I external references, performing the *javah* step is not necessary for PL/I native methods.

The complete procedure statement for the sample program looks like this:

```
 Java_callingPLI_callToPLI:
 Proc( JNIEnv , MyJObject )
   External( "Java_callingPLI_callToPLI" )
   Options( NoDescriptor ByValue );
```

JNI Include File: The two PL/I include files which contain the PL/I definitions of the Java Native interface are *ibmzjni.inc* which in turn includes *ibmzjnim.inc*. These include files are included with this statement:

```
   %include ibmzjni;
```

The *ibmzjni* and *ibmzjnim* include files are provided in the PL/I **SIBMZSAM** data set.

The Complete PL/I Procedure: For completeness, here is the entire PL/I program that defines the native method:

```
 *Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
 *Process Display(Std) Dllinit Extrn(Short);
 *Process Default( ASCII IEEE );
 PliJava_Demo: Package Exports(*);

  Java_callingPLI_callToPLI:
  Proc( JNIEnv , MyJObject )
```

```
  External( "Java_callingPLI_callToPLI" )
  Options( NoDescriptor ByValue );

%include ibmzjni;
Dcl myJObject        Type jObject;

Display('Hello from Enterprise PL/I!');

End;
```

Step 4: Compiling and Linking the PL/I Program

Compiling the PL/I Program: Compile the PL/I sample program with the
following command:

```
pli -c hiFromPLI.pli
```

Linking the Shared Library: Link the resulting PL/I object deck into a shared
library with this command:

```
c89 -o libhiFromPLI.so hiFromPLI.o
```

Be sure to include the *lib* prefix on the name of the PL/I shared library or the Java
class loader will not find it.

Step 5: Running the Sample Program
Run the Java - PL/I sample program with this command:

```
java callingPLI
```

The output of the sample program will look like this:

```
Hello from Enterprise PL/I!
And Hello from Java, too!
```

The first line written from the PL/I native method. The second line is from the
calling Java class after returning from the PL/I native method call.

JNI Sample Program #2 - Passing a String

Writing Java Sample Program #2

This sample program passes a string back and forth between Java and PL/I. Refer
to Figure 82 on page 324 for the complete listing of the *jPassString.java* program.
The Java portion has one Java class, *jPassString.java*. The native method, written in
PL/I, is contained in *passString.pli*. Much of the information from the first sample
program applies to this sample program as well. Only new or different aspects will
be discussed for this sample program.

Step 1: Writing the Java Program

Declare the Native Method: The native method for this sample program looks
like this:

```
public native void pliShowString();
```

Load the Native Library: The Java statement that loads the native library for this
sample program looks like this:

```
static {
  System.loadLibrary("passString");
}
```

Write the Java Main Method: The *jPassString* class also includes a *main* method to instantiate the class and call the native method. The *main* method instantiates *jPassString* and calls the *pliShowString()* native method.

This sample program prompts the user for a string and reads that value in from the command line. This is done within a *try/catch* statement as shown in Figure 82.

```
// Read a string, call PL/I, display new string upon return
import java.io.*;

public class jPassString{

 /* Field to hold Java string                            */
 String myString;

 /* Load the PL/I native library                         */
 static {
       System.loadLibrary("passString");
 }

 /* Declare the PL/I native method                       */
 public native void pliShowString();

 /* Main Java class                                      */
 public static void main(String[] arg) {

   System.out.println(" ");

   /* Instantiate Java class and initialize string       */
   jPassString myPassString = new jPassString();
   myPassString.myString = " ";

   /* Prompt user for a string                           */
   try {
       BufferedReader in = new BufferedReader(
         new InputStreamReader(System.in));

       /* Process until 'quit' received               */
       while (!myPassString.myString.equalsIgnoreCase("quit")) {
         System.out.println(
           "From Java: Enter a string or 'quit' to quit.");
         System.out.print("Java Prompt > ");
         /* Get string from command line               */
         myPassString.myString = in.readLine();
         if (!myPassString.myString.equalsIgnoreCase("quit"))
           {
            /* Call PL/I native method                 */
            myPassString.pliShowString();
            /* Return from PL/I and display new string */
            System.out.println(" ");
            System.out.println(
                "From Java: String set by PL/I is: "
              + myPassString.myString );
           }
       }
     } catch (IOException e) {
       }
 }
}
```

Figure 82. Java Sample Program #2 - Passing a String

Step 2: Compiling the Java Program

The command to compile the Java code would look like this:

```
javac jPassString.java
```

Step 3: Writing the PL/I Program

All of the information about writing the PL/I ″Hello World″ sample program applies to this program as well.

Correct Form of PL/I Procedure Name and Procedure Statement: The PL/I procedure name for this program would be *Java_jPassString_pliShowString*.

The complete procedure statement for the sample program looks like this:

```
Java_jPassString_pliShowString:
Proc( JNIEnv , myjobject )
  external( "Java_jPassString_pliShowString" )
  options( byvalue nodescriptor );
```

JNI Include File: The two PL/I include files which contain the PL/I definitions of the Java Native interface are *ibmzjni.inc* which in turn includes *ibmzjnim.inc*. These include files are included with this statement:

```
%include ibmzjni;
```

The *ibmzjni* and *ibmzjnim* include files are provided in the PL/I **SIBMZSAM** data set.

The Complete PL/I Procedure: The complete PL/I program is shown in Figure 83 on page 326. This sample PL/I program makes several calls through the JNI.

Upon entry, a reference to the calling Java Object, *myObject* is passed into the PL/I procedure. The PL/I program will use this reference to get information from the calling object. The first piece of information is the Class of the calling object which is retrieved using the *GetObjectClass* JNI function. This Class value is then used by the *GetFieldID* JNI function to get the identity of the Java string field in the Java object that we are interested in. This Java field is further identified by providing the name of the field, *myString*, and the JNI field descriptor, *Ljava/lang/String;*, which identifies the field as a Java String field. The value of the Java string field is then retrieved using the *GetObjectField* JNI function. Before PL/I can use the Java string value, it must be unpacked into a form that PL/I can understand. The *GetStringUTFChars* JNI function is used to convert the Java string into a PL/I varyingz string which is then displayed by the PL/I program.

After displaying the retrieved Java string, the PL/I program prompts the user for a PL/I string to be used to update the string field in the calling Java object. The PL/I string value is converted to a Java string using the *NewString* JNI function. This new Java string is then used to update the string field in the calling Java object using the *SetObjectField* JNI function.

When the PL/I program ends control is returned to Java, where the newly updated Java string is displayed by the Java program.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Default( ASCII IEEE );
plijava_demo: package exports(*);

 Java_passString_pliShowString:
 Proc( JNIEnv , myJObject )
   external( "Java_jPassString_pliShowString" )
   options( byvalue nodescriptor );

  %include ibmzjni;

  Dcl myBool          Type jBoolean;
  Dcl myClazz         Type jClass;
  Dcl myFID           Type jFieldID;
  Dcl myJObject       Type jObject;
  Dcl myJString       Type jString;
  Dcl newJString      Type jString;
  Dcl myID            Char(9)   Varz static init( 'myString' );
  Dcl mySig           Char(18)  Varz static
                      init( 'Ljava/lang/String;' );
  Dcl pliStr          Char(132) Varz Based(pliStrPtr);
  Dcl pliReply        Char(132) Varz;
  Dcl pliStrPtr       Pointer;
  Dcl nullPtr         Pointer;

  Display(' ');

  /* Get information about the calling Class            */
  myClazz = GetObjectClass(JNIEnv, myJObject);

  /* Get Field ID for String field from Java           */
  myFID = GetFieldID(JNIEnv, myClazz, myID, mySig );

  /* Get the Java String in the string field           */
  myJString = GetObjectField(JNIEnv, myJObject, myFID );

  /* Convert the Java String to a PL/I string          */
  pliStrPtr = GetStringUTFChars(JNIEnv, myJString, myBool );

  Display('From  PLI: String retrieved from Java is: ' || pliStr );
  Display('From  PLI: Enter a string to be returned to Java:' )
         reply(pliReply);

  /* Convert the new PL/I string to a Java String        */
  newJString = NewString(JNIEnv, trim(pliReply), length(pliReply) );

  /* Change the Java String field to the new string value */
  nullPtr = SetObjectField(JNIEnv, myJObject, myFID, newJString);

 End;

end;
```

Figure 83. PL/I Sample Program #2 - Passing a String

Step 4: Compiling and Linking the PL/I Program

Compiling the PL/I Program: Compile the PL/I sample program with the
following command:

```
pli -c passString.pli
```

Linking the Shared Library: Link the resulting PL/I object deck into a shared library with this command:

```
c89 -o libpassString.so passString.o
```

Be sure to include the *lib* prefix on the name or the PL/I shared library or the Java class loader will not find it.

Step 5: Running the Sample Program

Run the Java - PL/I sample program with this command:

```
java jPassString
```

The output of the sample program, complete with the prompts for user input from both Java and PL/I, will look like this:

```
>java jPassString

From Java: Enter a string or 'quit' to quit.
Java Prompt > A string entered in Java

From  PLI: String retrieved from Java is: A string entered in Java
From  PLI: Enter a string to be returned to Java:
A string entered in PL/I

From Java: String set by PL/I is: A string entered in PL/I
From Java: Enter a string or 'quit' to quit.
Java Prompt > quit
>
```

JNI Sample Program #3 - Passing an Integer

Writing Java Sample Program #3

This sample program passes an integer back and forth between Java and PL/I. Refer to Figure 84 on page 328 for the complete listing of the *jPassInt.java* program. The Java portion has one Java class, *jPassInt.java*. The native method, written in PL/I, is contained in *passInt.pli*. Much of the information from the first sample program applies to this sample program as well. Only new or different aspects will be discussed for this sample program.

Step 1: Writing the Java Program

Declare the Native Method: The native method for this sample program looks like this:

```
public native void pliShowInt();
```

Load the Native Library: The Java statement that loads the native library for this sample program looks like this:

```
static {
  System.loadLibrary("passInt");
}
```

Write the Java Main Method: The *jPassInt* class also includes a *main* method to instantiate the class and call the native method. The *main* method instantiates *jPassInt* and calls the *pliShowInt()* native method.

This sample program prompts the user for an integer and reads that value in from the command line. This is done within a *try/catch* statement as shown in Figure 84 on page 328.

```
// Read an integer, call PL/I, display new integer upon return
import java.io.*;
import java.lang.*;

public class jPassInt{

 /* Fields to hold Java string and int                */
 int myInt;
 String myString;

 /* Load the PL/I native library                      */
 static {
        System.loadLibrary("passInt");
 }

 /* Declare the PL/I native method                    */
 public native void pliShowInt();

 /* Main Java class                                   */
 public static void main(String[] arg) {

   System.out.println(" ");

   /* Instantiate Java class and initialize string    */
   jPassInt pInt = new jPassInt();
   pInt.myInt = 1024;
   pInt.myString = " ";

   /* Prompt user for an integer                       */
   try {
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));

        /* Process until 'quit' received               */
        while (!pInt.myString.equalsIgnoreCase("quit")) {
          System.out.println
            ("From Java: Enter an Integer or 'quit' to quit.");
          System.out.print("Java Prompt > ");
          /* Get string from command line              */
          pInt.myString = in.readLine();
          if (!pInt.myString.equalsIgnoreCase("quit"))
            {
             /* Set int to integer value of String      */
             pInt.myInt = Integer.parseInt( pInt.myString );
             /* Call PL/I native method                 */
             pInt.pliShowInt();
             /* Return from PL/I and display new string */
             System.out.println(" ");
             System.out.println
               ("From Java: Integer set by PL/I is: " + pInt.myInt );
            }
        }
     }  catch (IOException e) {
        }
 }

}
```

Figure 84. Java Sample Program #3 - Passing an Integer

Step 2: Compiling the Java Program

The command to compile the Java code would look like this:

```
 javac jPassInt.java
```

Step 3: Writing the PL/I Program

All of the information about writing the PL/I ″Hello World″ sample program applies to this program as well.

Correct Form of PL/I Procedure Name and Procedure Statement: The PL/I procedure name for this program would be *Java_jPassInt_pliShowInt.*

The complete procedure statement for the sample program looks like this:

```
 Java_passNum_pliShowInt:
 Proc( JNIEnv , myjobject )
   external( "Java_jPassInt_pliShowInt" )
   options( byvalue nodescriptor );
```

JNI Include File: The two PL/I include files which contain the PL/I definitions of the Java Native interface are *ibmzjni.inc* which in turn includes *ibmzjnim.inc.* These include files are included with this statement:

```
   %include ibmzjni;
```

The *ibmzjni* and *ibmzjnim* include files are provided in the PL/I **SIBMZSAM** data set.

The Complete PL/I Procedure: The complete PL/I program is shown in Figure 85 on page 330. This sample PL/I program makes several calls through the JNI.

Upon entry, a reference to the calling Java object, *myObject*, is passed into the PL/I procedure. The PL/I program will use this reference to get information from the calling object. The first piece of information is the Class of the calling object which is retrieved using the *GetObjectClass* JNI function. This Class value is then used by the *GetFieldID* JNI function to get the identity of the Java integer field in the Java object that we are interested in. This Java field is further identified by providing the name of the field, *myInt*, and the JNI field descriptor, *I*, which identifies the field as an integer field. The value of the Java integer field is then retrieved using the *GetIntField* JNI function which is then displayed by the PL/I program.

After displaying the retrieved Java integer, the PL/I program prompts the user for a PL/I integer to be used to update the integer field in the calling Java object. The PL/I integer value is then used to update the integer field in the calling Java object using the *SetIntField* JNI function.

When the PL/I program ends, control is returned to Java, where the newly updated Java integer is displayed by the Java program.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Default( ASCII IEEE );
plijava_demo: package exports(*);

 Java_passNum_pliShowInt:
 Proc( JNIEnv , myjobject )
   external( "Java_jPassInt_pliShowInt" )
   options( byvalue nodescriptor );

  %include ibmzjni;

  Dcl myClazz        Type jClass;
  Dcl myFID          Type jFieldID;
  Dcl myJInt         Type jInt;
  dcl rtnJInt        Type jInt;
  Dcl myJObject      Type jObject;
  Dcl pliReply       Char(132) Varz;
  Dcl nullPtr        Pointer;

  Display(' ');

  /* Get information about the calling Class          */
  myClazz = GetObjectClass(JNIEnv, myJObject);

  /* Get Field ID for int field from Java             */
  myFID = GetFieldID(JNIEnv, myClazz, "myInt", "I");

  /* Get Integer value from Java                      */
  myJInt = GetIntField(JNIEnv, myJObject, myFID);

  display('From  PLI: Integer retrieved from Java is: ' || trim(myJInt) );
  display('From  PLI: Enter an integer to be returned to Java:' )
         reply(pliReply);

  rtnJInt = pliReply;

  /* Set Integer value in Java from PL/I              */
  nullPtr = SetIntField(JNIEnv, myJObject, myFID, rtnJInt);

 End;

end;
```

Figure 85. PL/I Sample Program #3 - Passing an Integer

Step 4: Compiling and Linking the PL/I Program

Compiling the PL/I Program: Compile the PL/I sample program with the following command:

```
pli -c passInt.pli
```

Linking the Shared Library: Link the resulting PL/I object deck into a shared library with this command:

```
c89 -o libpassInt.so passInt.o
```

Be sure to include the *lib* prefix on the name or the PL/I shared library or the Java class loader will not find it.

Step 5: Running the Sample Program
Run the Java - PL/I sample program with this command:

```
java jPassInt
```

The output of the sample program, complete with the prompts for user input from both Java and PL/I, will look like this:

```
>java jPassInt

From Java: Enter an Integer or 'quit' to quit.
Java Prompt > 12345

From  PLI: Integer retrieved from Java is: 12345
From  PLI: Enter an integer to be returned to Java:
54321

From Java: Integer set by PL/I is: 54321
From Java: Enter an Integer or 'quit' to quit.
Java Prompt > quit
>
```

JNI Sample Program #4 - Java Invocation API

Writing Java Sample Program #4

This sample program is a little different from the previous samples. In this sample PL/I invokes Java first, through the Java Invocation API, creating an embedded Java Virtual Machine (JVM). PL/I then calls a Java method passing it a string which the Java method then displays.

The PL/I sample program is named *createJVM.pli* and the Java method it calls is contained in *javaPart.java*.

Step 1: Writing the Java Program
Since this sample does not use a PL/I native method there is no need to declare one. Instead, the Java portion for this sample is just a simple Java method.

Write the Java Main Method: The *javaPart* class contains only one statement. This statement prints out a short 'Hello World...' from Java then appends the string that was passed to it from the PL/I program. The entire class is shown in Figure 86.

```
// Receive a string from PL/I then display it after saying "Hello"
public class javaPart {
   public static void main(String[] args) {
      System.out.println("From Java  -  Hello World... " + args[0]);
   }
}
```

Figure 86. Java Sample Program #4 - Receiving and printing a String

Step 2: Compiling the Java Program
The command to compile the Java code looks like this:

```
javac javaPart.java
```

Step 3: Writing the PL/I Program
Most of the information about writing the PL/I ″Hello World″ sample program applies to this program as well. However, since in this sample PL/I is calling Java, there are some additional points to consider.

Correct Form of PL/I Procedure Name and Procedure Statement: Since in this sample the PL/I program is calling Java, the PL/I program is MAIN. There is no need to be concerned about the external name of this PL/I program because it is not referenced.

The complete procedure statement for the sample program looks like this:

```
createJVM: Proc Options(Main);
```

JNI Include File: The two PL/I include files which contain the PL/I definitions of the Java Native interface are *ibmzjni.inc* which in turn includes *ibmzjnim.inc*. Even though in this sample PL/I is calling Java, these include files are still necessary. These include files are included with this statement:

```
   %include ibmzjni;
```

The *ibmzjni* and *ibmzjnim* include files are provided in the PL/I **SIBMZSAM** data set.

Linking the PL/I program with the Java library: Since this PL/I sample program calls Java we need to prepare it to link to the Java library. The Java libraries are linked with XPLINK and the PL/I modules are not. PL/I can still link to and call XPLINK libraries but must use the PLIXOPT variable to specify the **XPLINK=ON** run-time option. This is what the declare of the PLIXOPT variable would look like:

```
   Dcl PLIXOPT    Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );
```

For a description of using PLIXOPT, see *z/OS Language Environment Programming Guide*.

Using the Java Invocation API: This PL/I sample program calls the Java Invocation API *JNI_CreateJavaVM* to create its own embedded JVM. This API requires certain structures to be set up and initialized correctly as shown in Figure 87 on page 334. First, *JNI_GetDefaultJavaVMInitArgs* is called to get the default initilization options. Next, these default options are modified with the addition of the *java.class.path* information. Finally, *JNI_CreateJavaVM* is called to create the embedded JVM.

The Complete PL/I Program: The complete PL/I program is shown in Figure 87 on page 334. This sample PL/I program makes several calls through the JNI.

In this sample the reference to the Java object, a newly created JVM in this case, is not passed in but is instead returned from the call to the *JNI_CreateJavaVM* API. The PL/I program will use this reference to get information from the JVM. The first piece of information is the Class containing the Java method we wish to call. This Class is found by using the *FindClass* JNI function. The Class value is then used by the *GetStaticMethodID* JNI function to get the identity of the Java method that will be called.

Before we call this Java method we need to convert the PL/I string into a format that Java understands. The PL/I program holds the string in ASCII format. Java strings are stored in UTF format. In addition, Java strings are not really strings as PL/I programmers understand them but are themselves a Java class and can only be modified through methods. We create a Java string barry by using the *NewStringUTF* JNI function. This function returns a Java object called *myJString* that contains the PL/I string converted to UTF. Next we create a Java object array by calling the *NewObjectArray* JNI function passing it the reference to the *myJString* object. This function returns a reference to a Java object array containing the string we want the Java method to display.

Now the Java method can be called with the *CallStaticVoidMethod* JNI function and will then display the string passed to it. After displaying the string, the PL/I program destroys the embedded JVM with the *DestroyJavaVM* JNI function and the PL/I program completes.

The complete source of the PL/I program is in Figure 87 on page 334.

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 );
*Process LangLVL( SAA2 ) Margins( 1, 100 ) ;
*Process Display(STD) Rent;
*Process Default( ASCII ) Or('|');
createJVM: Proc Options(Main);

  %include ibmzjni;

  Dcl myJObjArray Type jobjectArray;
  Dcl myClass     Type jclass;
  Dcl myMethodID  Type jmethodID;
  Dcl myJString   Type jstring;
  Dcl myRC        Fixed Bin(31)  Init(0);
  Dcl myPLIStr    Char(50) Varz
                  Init('... a PLI string in a Java Virtual Machine!');
  Dcl OptStr      Char(1024) Varz;
  Dcl myNull      Pointer;
  Dcl VM_Args     Like JavaVMInitArgs;
  Dcl myOptions   Like JavaVMOption;
  Dcl PLIXOPT     Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );

  Display('From PL/I  - Beginning execution of createJVM...');
  /* Important difference between J1.3 and J1.4:            */
  /* J1.3 requires VM_Args.version = JNI_VERSION_1_2;       */
  /* J1.4 requires VM_Args.version = JNI_VERSION_1_4;       */
  VM_Args.version      = JNI_VERSION_1_4;

  myRC =  JNI_GetDefaultJavaVMInitArgs( addr(VM_Args) );
  OptStr            = "-Djava.class.path=.:";
  myOptions.theOptions = addr(OptStr);
  VM_Args.nOptions     = 1;
  VM_Args.JavaVMOption = addr(myOptions);

  /* Create the Java VM                                     */
  myrc = JNI_CreateJavaVM(
            addr(JavaVM),
            addr(JNIEnv),
            addr(VM_Args) );

  /* Get the Java Class for the javaPart class             */
  myClass = FindClass(JNIEnv, "javaPart");
  /* Get static method ID                                  */
  myMethodID = GetStaticMethodID(JNIEnv, myClass,"main",
            "([Ljava/lang/String;)V" );
  /* Create a Java String Object from the PL/I string.     */
  myJString = NewStringUTF(JNIenv, myPLIStr);
  myJObjArray = NewObjectArray(JNIEnv, 1,
              FindClass(JNIEnv,"java/lang/String"), myJString);
  Display('From PL/I  - Calling Java method in new JVM from PL/I...');
  Display(' ');
  myNull = CallStaticVoidMethod(JNIEnv, myClass,
            myMethodID, myJObjArray );
  /* destroy the Java VM                                    */
  Display(' ');
  Display('From PL/I  - Destroying the new JVM from PL/I...');
  myRC = DestroyJavaVM( JavaVM ); /* rc = -1 is OK          */
end;
```

Figure 87. PL/I Sample Program #4 - Calling the Java Invocation API

Step 4: Compiling and Linking the PL/I Program

Compiling the PL/I Program: Compile the PL/I sample program with the following command:

```
pli -c createJVM.pli
```

Linking the PL/I program to the Java Library: Link the resulting PL/I object deck with the Java library with this command:

```
c89 -o createJVM createJVM.o $JAVA_HOME/bin/classic/libjvm.x
```

Notice the reference to the **$JAVA_HOME** environment variable. This variable should point to the directory that your Java 1.4 product is installed in. For example, to set up this variable in your environment you would use the following command:

```
export JAVA_HOME="/usr/lpp/java/J1.4"
```

In this case the Java 1.4 product is assumed to be installed in */usr/lpp/java/J1.4*.

Step 5: Running the Sample Program

Run the PL/I - Java sample program with this command:

```
createJVM
```

The output of the sample program will look like this:

```
From PL/I  - Beginning execution of createJVM...
From PL/I  - Calling Java method in new JVM from PL/I...
From Java  - Hello World... ... a PLI string in a Java Virtual Machine!
From PL/I  - Destroying the new JVM from PL/I...
```

Determining equivalent Java and PL/I data types

When you communicate with Java from PL/I you will need to match the data types between the two programming languages. This table shows Java primitive types and their PL/I equivalents:

Table 32. Java Primitive Types and PL/I Native Equivalents

Java Type	PL/I Type	Size in Bits
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	21
double	jdouble	53
void	jvoid	n/a

Part 5. Specialized programming tasks

Chapter 15. 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 XML support does not provide XML generation, which must instead be accomplished by PL/I program logic or by using the XMLCHAR built-in function.

The XML support has no special environmental requirements. It executes in all the principal run-time environments, including CICS, IMS, and MQ Series, as well as z/OS batch and TSO.

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.

For each parser event, you must provide a PL/I function that accepts the
appropriate parameters and returns the appropriate return value - as in the
example code below. Note in particular that the return value must be returned
BYVALUE. Also, these functions must all use the OPTLINK linkage. You can use
the DEFAULT(LINKAGE(OPTLINK)) option to specify this linkage, or you can
specify it on the individual PROCEDUREs and ENTRYs via the
OPTIONS(LINKAGE(OPTLINK)) attribute.

The PLISAXA built-in subroutine

The PLISAXA built-in subroutine allows you to invoke the XML parser for an XML
document residing in a buffer in your program.

```
►►──PLISAXA(e,p,x,n─────────)────────────────────────────────►◄
                      └─,c─┘
```

e   An event structure

p   A pointer value or ″token″ that the parser will pass back to the event functions

x   The address of the buffer containing the input XML

n   The number of bytes of data in that buffer

c   A numeric expression specifying the purported codepage of that XML

Note that if the XML is contained in a CHARACTER VARYING or a WIDECHAR
VARYING string, then the ADDRDATA built-in function should be used to obtain
the address of the first data byte.

Also note that if the XML is contained in a WIDECHAR string, the value for the
number of bytes is twice the value returned by the LENGTH built-in function.

The PLISAXB built-in subroutine

The PLISAXB built-in subroutine allows you to invoke the XML parser for an XML
document residing in a file.

```
►►──PLISAXB(e,p,x───────)──────────────────────────────────────►◄
                └─,c─┘
```

e   An event structure

p   A pointer value or ″token″ that the parser will pass back to the event functions

x   A character string expression specifying the input file

c   A numeric expression specifying the purported codepage of that XML

Under batch, the character string specifying the input file should have the form
'file://dd:ddname', where ddname is the name of the DD statement specifying the
file.

Under z/OS UNIX, the character string specifying the input file should have the
form 'file://filename', where filenameis the name of a z/OS UNIX file.

Under both batch and z/OS UNIX, the character string specifying the input file
should have no leading or trailing blanks.

The input XML file must be less than 2G in size. Moreover, since the parser will read the entire file into memory, the REGION for your program must be large enough for the parser to obtain a piece of storage that can contain all of the document.

The SAX event structure

The event structure is a structure consisting of 24 LIMITED ENTRY variables which point to functions that the parser will invoke for various ″events″.

All these ENTRYs must use the OPTLINK linkage.

The descriptions below of each event refer to the example of an XML document in Figure 88. In these descriptions, the term ″XML text″ refers to the string based on the pointer and length passed to the event.

```
xmlDocument =
    '<?xml version="1.0" standalone="yes"?>'
 || '<!--This document is just an example-->'
 || '<sandwich>'
 || '<bread type="baker&quot;s best"/>'
 || '<?spread please use real mayonnaise ?>'
 || '<meat>Ham &amp; turkey</meat>'
 || '<filling>Cheese, lettuce, tomato, etc.</filling>'
 || '<![CDATA[We should add a <relish> element in future!]]>'
 || '</sandwich>'
 || 'junk';
```

Figure 88. 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
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
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 linkage(optlink) );

  define alias event_end_of_document
      limited entry( pointer )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_predefined_ref
      limited entry( pointer, char(1) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) nodescriptor );

  define alias event_character_ref
      limited entry( pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_pi
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );

  define alias event_exception
      limited entry( pointer, pointer, fixed bin(31),
                     fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue linkage(optlink) );
```

Figure 89. 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 90. PLISAXA coding example - event structure

```
dcl token      char(8);

dcl xmlDocument char(4000) var;

xmlDocument =
   '<?xml version="1.0" standalone="yes"?>'
|| '<!--This document is just an example-->'
|| '<sandwich>'
|| '<bread type="baker&quot;s best"/>'
|| '<?spread please use real mayonnaise ?>'
|| '<meat>Ham &amp; turkey</meat>'
|| '<filling>Cheese, lettuce, tomato, etc.</filling>'
|| '<![CDATA[We should add a <relish> element in future!]]>'.
|| '</sandwich>'
|| 'junk';


call plisaxa( eventHandler,
              addr(token),
              addrdata(xmlDocument),
              length(xmlDocument) );

end;
```

Figure 91. PLISAXA coding example - main routine

```
dcl chars char(32000) based;

start_of_document:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' length=' || tokenlength );

  return(0);
end;

version_information:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

encoding_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 1 of 8)

```
standalone_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken     pointer;
  dcl xmlToken      pointer;
  dcl tokenLength   fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

document_type_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken     pointer;
  dcl xmlToken      pointer;
  dcl tokenLength   fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

end_of_document:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken     pointer;

  put skip list( lowercase( procname() ) );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 2 of 8)

```
start_of_element:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

attribute_name:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

attribute_characters:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 3 of 8)

```
attribute_predefined_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) nodescriptor );

  dcl userToken      pointer;
  dcl reference    char(1);

  put skip list( lowercase( procname() )
   || ' ' || hex(reference ) );

  return(0);
end;

attribute_character_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken       pointer;
  dcl reference       fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || hex(reference ) );

  return(0);
end;

end_of_element:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken       pointer;
  dcl xmlToken        pointer;
  dcl tokenLength     fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 4 of 8)

```
start_of_CDATA:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

end_of_CDATA:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

content_characters:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 5 of 8)

```
content_predefined_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) nodescriptor );

  dcl userToken     pointer;
  dcl reference    char(1);

  put skip list( lowercase( procname() )
   || ' ' || hex(reference ) );

  return(0);
end;

content_character_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl reference      fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || hex(reference ) );

  return(0);
end;

processing_instruction:
  proc( userToken, piTarget, piTargetLength,
                          piData, piDataLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl piTarget       pointer;
  dcl piTargetLength fixed bin(31);
  dcl piData         pointer;
  dcl piDataLength   fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 6 of 8)

```
comment:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

unknown_attribute_reference:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

unknown_content_reference:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 92. PLISAXA coding example - event routines (Part 7 of 8)

```
  start_of_prefix_mapping:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  end_of_prefix_mapping:
    proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
     || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

    return(0);
  end;

  exception:
    proc( userToken, xmlToken, currentOffset, errorID )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl currentOffset  fixed bin(31);
    dcl errorID        fixed bin(31);

    put skip list( lowercase( procname() )
     || ' errorid =' || errorid );

    return(0);
  end;
end;
```

Figure 92. 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 93. 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 33. 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 33. 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.
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.

Table 33. Continuable Exceptions (continued)

Number	Description	Parser Action on Continuation
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. **NOTE:** The PL/I XML parser will deviate from the standard and accept ″<″ as valid in an attribute string.
50	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
51	The document was encoded in EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
52	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an ASCII encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
53	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
54	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
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.

Table 33. Continuable Exceptions (continued)

Number	Description	Parser Action on Continuation
56	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
57	The document was encoded in ASCII, and the document encoding declaration specified a supported ASCII encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
58	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported EBCDIC encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
59	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
60	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
61	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
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 34. 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.

Table 34. Terminating Exceptions (continued)

Number	Description
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.
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 ':'.

Table 34. Terminating Exceptions (continued)

Number	Description
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.
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.

Table 34. Terminating Exceptions (continued)

Number	Description
306	The document was encoded in ASCII, but the CODEPAGE compiler option specified a supported EBCDIC code page.
307	The document was encoded in ASCII, but the CODEPAGE compiler option specified Unicode.
308	The document was encoded in ASCII, but the CODEPAGE compiler option did not specify a supported EBCDIC code page, ASCII or Unicode.
309	The CODEPAGE compiler option specified a supported ASCII code page, but the document was encoded in Unicode.
310	The CODEPAGE compiler option specified a supported EBCDIC code page, but the document was encoded in Unicode.
311	The CODEPAGE compiler option specified an unsupported code page, but the document was encoded in Unicode.
312	The document was encoded in ASCII, but both the 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.
501, 502, 503	An internal error occurred in PLISAX(A|B). Please contact IBM support.
510	Memory allocation failed for the PLISAXA internal data structures. Increase the amount of storage available to the application program.
520	Memory allocation failed for the PLISAXB internal data structures. Increase the amount of storage available to the application program.
521	An internal error occurred in PLISAX(A|B). Please contact IBM support.
523	PLISAXB encountered a file I/O error.
524	Memory allocation failed in PLISAXB while attempting to cache the XML document from the file system. Increase the amount of storage available to the application program.
525	An unsupported URI scheme was specified to PLISAXB.
526	The XML document provided to PLISAXB was less than the minimum of 4 characters or was too large.
527, 560	An internal error occurred in PLISAX(A|B). Please contact IBM support.
561	No event handler was specified to either PLISAX(A|B).
562, 563, 580, 581	An internal error occurred in PLISAX(A|B). Please contact IBM support.
600 to 99,999	Internal error. Please report the error to your service representative.

Chapter 16. Using PLIDUMP

This section provides information about dump options and the syntax used to call PLIDUMP, and describes PL/I-specific information included in the dump that can help you debug your routine.

Note: PLIDUMP conforms to National Language Support standards.

Figure 94 shows an example of a PL/I routine calling PLIDUMP to produce a z/OS Language Environment dump. In this example, the main routine PLIDMP calls PLIDMPA, which then calls PLIDMPB. The call to PLIDUMP is made in routine PLIDMPB.

```
%PROCESS MAP GOSTMT SOURCE STG LIST OFFSET LC(101);
 PLIDMP: PROC  OPTIONS(MAIN) ;

   Declare   (H,I) Fixed bin(31) Auto;
   Declare   Names Char(17) Static init('Bob Teri Bo Jason');
   H = 5;  I = 9;
   Put skip list('PLIDMP Starting');
   Call PLIDMPA;

     PLIDMPA:  PROC;
       Declare  (a,b)  Fixed bin(31)   Auto;
       a = 1;  b = 3;
       Put skip list('PLIDMPA Starting');
       Call PLIDMPB;

         PLIDMPB:  PROC;
           Declare  1 Name  auto,
                      2  First   Char(12) Varying,
                      2  Last    Char(12) Varying;
           First = 'Teri';
           Last  = 'Gillispy';
           Put skip list('PLIDMPB Starting');
           Call PLIDUMP('TBFC','PLIDUMP called from procedure PLIDMPB');
           Put Data;
         End PLIDMPB;

     End PLIDMPA;

 End PLIDMP;
```

Figure 94. Example PL/I routine calling PLIDUMP

The syntax and options for PLIDUMP are shown below.

►►──PLIDUMP──(*character-string-expression 1*,*character-string-expression 2*)──────────►◄

character-string-expression 1
>   is a dump options character string consisting of one or more of the following:
>
>   **A**   Requests information relevant to all tasks in a multitasking program.
>
>   **B**   BLOCKS (PL/I hexadecimal dump).
>
>   **C**   Continue. The routine continues after the dump.
>
>   **E**   Exit from current task of a multitasking program. Program continues to run after requested dump is completed.
>
>   **F**   FILES.

H	STORAGE.

This includes all Language Environment storage, and hence all the BASED and CONTROLLED storage acquired via ALLOCATE statements.

> **Note:** A ddname of CEESNAP should be specified with the H option to produce a SNAP dump of a PL/I routine, but if this is omitted, Language Environment will issue a message but still produce a dump with much very useful information.

K	BLOCKS (when running under CICS). The Transaction Work Area is included.
NB	NOBLOCKS.
NF	NOFILES.
NH	NOSTORAGE.
NK	NOBLOCKS (when running under CICS).
NT	NOTRACEBACK.
O	Only information relevant to the current task in a multitasking program.
S	Stop. The enclave is terminated with a dump.
T	TRACEBACK.

T, F, and C are the default options.

character-string-expression 2
is a user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP usage notes

If you use PLIDUMP, the following considerations apply:
- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- A DD statement with the ddname PLIDUMP, PL1DUMP, or CEEDUMP can be used to define the data set for the dump.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 133 to prevent dump records from wrapping. If SYSOUT is used as the target in any one of these DDs, you must specify MSGFILE(SYSOUT,FBA,133,0) or MSGFILE(SYSOUT,VBA,137,0) to ensure that the lines are not wrapped.
- When you specify the H option in a call to PLIDUMP, the PL/I library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.
- Support for SNAP dumps using PLIDUMP is only provided under z/OS. SNAP dumps are not produced in a CICS environment.
  - If the SNAP is not successful, the CEE3DMP DUMP file displays the message:
    ```
    Snap was unsuccessful
    ```
  - If the SNAP is successful, CEE3DMP displays the message:

```
Snap was successful; snap ID = nnn
```

where *nnn* corresponds to the SNAP identifier described above. An unsuccessful SNAP does not increment the identifier.

- If you want the program unit name, program unit address and program unit offset to be listed correctly in the dump traceback table, you need to make sure that your PL/I program unit is compiled with a compile-time option other than TEST(NONE,NOSYM). For example, you can specify the option as TEST(NOSYM,NOHOOK,BLOCK).

If you want to ensure portability across system platforms, use PLIDUMP to generate a dump of your PL/I routine.

Locating variables in the PLIDUMP output

To find variables in the PLIDUMP output, you should compile your program with the MAP option. The MAP option will cause the compiler to add to the listing a table showing the offset within AUTOMATIC and STATIC storage of all level-1 variables that are AUTOMATIC or STATIC.

To find a variable that is an element in a structure, it is also useful if you compile your program with the AGGREGATE option. This option will cause the compiler to add to the listing a table showing the offsets of all the elements of all the structures in your program.

Locating AUTOMATIC variables

To find an AUTOMATIC variable in the dump, you should find its offset within automatic using the output from the MAP option (and if necessary the AGGREGATE option). If PLIDUMP has been invoked with the B option, the dump output will contain a hex dump of the "Dynamic save area" (or DSA) for each block. This is the automatic storage for that block.

For example, consider the following simple program:

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31);
     5.0        dcl b fixed bin(31);
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBC');
    10.0          end;
    11.0
    12.0        a = 0;
    13.0        b = 29;
    14.0        b = 17 / a;
```

The result of the compiler MAP option for this program looks like this, except that there is actually one more column on the right and the columns are actually spaced much further apart:

```
* * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES


 A            1-0:4      Class = automatic,  Location = 160 : 0xA0(r13),
 B            1-0:5      Class = automatic,  Location = 164 : 0xA4(r13),
```

So, A is located at hex A0 off of register 13 and B is located at hex A4 off of register 13, where register 13 points to the DSA.

Since in this program PLIDUMP is called with the B option, it will include a hexadecimal dump of automatic storage for each block in the current calling chain. This will look like (again with the right columns cutoff):

```
Dynamic save area (TEST): 0AD963C8
     +000000 0AD963C8   10000000 0AD96188 00000000 00000000
     +000020 0AD963E8   00000000 00000000 00000000 00000000
     +000040 0AD96408   00000000 00000000 00000000 0AD96518
     +000060 0AD96428   00000000 00000000 00000000 00000000
     +000080 0AD96448 - +00009F 0AD96467   same as above
     +0000A0 0AD96468   00000000 0000001D 00100000 00000000
     +0000C0 0AD96488   0B300000 0A700930 0AD963C8 00000000
     +0000E0 0AD964A8   00000000 00000000 00000000 00000000
     +000100 0AD964C8   0AA47810 0A70E6D0 0AD96540 0AD960F0
     +000120 0AD964E8   00000001 0A70F4F8 0AD96318 00000000
     +000140 0AD96508   00000000 00000000 00000000 00000000
```

Since A is at hex offset A0 and B is at hex offset A4 in AUTOMATIC, the dump shows that A and B have the (expected) hex values of 00000000 and 0000001D respectively.

Please note that under the compiler options OPT(2) and OPT(3), some variables, particularly FIXED BIN and POINTER scalar variables, may never be allocated storage and thus could not be found in the dump output.

Locating STATIC variables

If you compiled your code with the RENT option, static variables are located in the WSA (the Writeable Static Area) for the current load module. The offset of a variable within the WSA can be found from the output of the MAP option, and the WSA is held in the Language Environment control block called the CAA. The value of the WSA is also listed in the Language Environment dump.

However, if you compiled your code with the NORENT option, EXTERNAL STATIC is found as usual (using the linker listing and the output of the compiler's MAP option). INTERNAL STATIC will be dumped as part of the Language Environment dump (if PLIDUMP was called with the B option).

Please also note that unlike the older PL/I compilers, the address of static is not dedicated to any one register.

For example, consider the program above with the variables changed to STATIC:

```
Compiler Source
  Line.File
    2.0      test: proc options(main);
    3.0
    4.0        dcl a fixed bin(31) static;
    5.0        dcl b fixed bin(31) static;
    6.0
    7.0        on error
    8.0          begin;
    9.0            call plidump('TFBC');
   10.0          end;
   11.0
   12.0        a = 0;
   13.0        b = 29;
   14.0        b = 17 / a;
```

When compiled with the NORENT option, the result of the compiler MAP option for this program looks like this, except that there is actually one more column on the right and the columns are actually spaced much further apart:

```
* * * * *  S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES


 A         1-0:4     Class = static,  Location = 0 : 0x0 + CSECT ***TEST2
 B         1-0:5     Class = static,  Location = 4 : 0x4 + CSECT ***TEST2
```

So, A is located at hex offset 00 into the static CSECT for the compilation unit TEST while B is located at hex offset 04.

Since in this program PLIDUMP is called with the B option, it will include a hexadecimal dump of static storage for each compilation in the current calling chain. This will look like (again with the right columns cutoff):

```
Static for procedure TEST    Timestamp: 2004.08.12
     +000000 0FC00AA0  00000000 0000001D 0FC00DC8 0FC00AC0
     +000020 0FC00AC0  0FC00AA8 00444042 00A3AE01 0FC009C8
     +000040 0FC00AE0  6E3BFFE0 00000000 00000000 00000000
     +000060 0FC00B00  00000000 00000000 00000000 00000000
     +000000 0AD963C8  10000000 0AD96188 00000000 00000000
```

So, A at hex offset 00 has the (expected) hex value 00000000, and B at hex offset 04 has the (also expected) hex value 0000001D or the decimal value 29.

Locating CONTROLLED variables

CONTROLLED variables are essentially LIFO stacks, and each CONTROLLED variable has an "anchor" that points to the top of that stack. The key to locating a CONTROLLED variable is to locate this anchor, and its location depends on the compiler options as described below.

In the rest of this discussion of CONTROLLED variables, the program source will be the same program as above, but with the storage class changed to CONTROLLED:

```
Compiler Source
  Line.File
     2.0      test: proc options(main);
     3.0
     4.0        dcl a fixed bin(31) controlled;
     5.0        dcl b fixed bin(31) controlled;
     6.0
     7.0        on error
     8.0          begin;
     9.0            call plidump('TFBHC');
    10.0          end;
    11.0
    12.0        allocate a, b;
    13.0        a = 0;
    14.0        b = 29;
    15.0        b = 17 / a;
```

Under NORENT WRITABLE
The result of the compiler MAP option looks like this, except that again there is actually one more column on the right and the columns are actually spaced much further apart:

```
* * * * *  S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES


 A         1-0:4     Class = static,  Location = 8 : 0x8 + CSECT ***TEST2
 B         1-0:5     Class = static,  Location = 12 : 0xC + CSECT ***TEST2
```

Note: that these lines describe the location of the anchors for A and B (not the location of A and B themselves). So, A's anchor is located at hex 08 into the static CSECT for the compilation unit TEST while B's anchor is located at hex 0C.

If PLIDUMP is called with the B option, it will include a hexadecimal dump of static storage for each compilation in the current calling chain. This will look like (again with the right columns cutoff):

```
Static for procedure TEST    Timestamp: . . .

  +000000 0FC00A88  0FC00DB0 0FC00AA8 102B8A30 102B8A50
  +000020 0FC00AA8  0FC00A88 00444042 00A3AE01 0FC009B0
  +000040 0FC00AC8  6E3BFFE0 00000000 00000000 00000000
```

So, A's anchor is at 102B8A30 and B's is at 102B8A50. But since these are CONTROLLED variables, their storage was obtained via ALLOCATE statements, and hence these addresses point into heap storage. But If PLIDUMP is called with the H option, it will include a hexadecimal dump of heap storage. This will look like (again with the right columns cutoff):

```
Enclave Storage:
  Initial (User) Heap
    +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000

      . . .
    +001A00 102B8A18  102B7018 00000020 0FC00A90 00000014
                      00000000 00000000 00000000 00000000
    +001A20 102B8A38  102B7018 00000020 0FC00A94 00000014
                      00000000 00000000 0000001D 00000000
```

Since A's anchor was at 102B8A30, A has the hex value 00000000, and since B's anchor was at 102B8A50, B has the (expected) hex value 0000001D.

Under NORENT NOWRITABLE(FWS)

The result of the compiler MAP option under these options looks like this, except that again there is actually one more column on the right and the columns are actually spaced much further apart:

```
 * * * * *  S T O R A G E  O F F S E T  L I S T I N G  * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 A         1-0:4      Class = automatic,  Location = 236 : 0xEC(r13)
 B         1-0:5      Class = automatic,  Location = 240 : 0xF0(r13)
```

Note: under these options, there is an extra level of indirection in locating CONTROLLED variables, and hence the lines above describe the locations of the addresses of the anchors for A and B. So, the address of A's anchor is located at hex EC into the automatic for the block TEST while B's is located at hex F0.

Since PLIDUMP is called with the B option, it will include a hexadecimal dump of automatic storage for each block in the current calling chain. This will look like (again with the right columns cutoff):

```
 Dynamic save area (TEST): 102973C8
  +000000 102973C8   10000000 10297188 00000000 8FC007DA
          ....
  +0000E0 102974A8  0FC00998 00000000 00000000 102B8A40
                    102B8A28 10297030 102977D0 8FDF3D7E
```

So, the address of A's anchor is 102B8A40 and B's is 102B8A28.

Since PLIDUMP was also called with the H option, it will include a hexadecimal dump of heap storage. This will look like (again with the right columns cutoff):

```
Enclave Storage:
   Initial (User) Heap
     +000000 102B7018  C8C1D5C3 0FC0F990 0FC0F990 00000000
        . . .
     +001A00 102B8A18  102B7018 00000018 00000000 0FC00A78
                       102B8A80 00000000 102B7018 00000018
     +001A20 102B8A38  102B8A20 0FC00A74 102B8A60 00000000
                       102B7018 00000020 102B8A40 00000014
     +001A40 102B8A58  00000000 00000000 00000000 00000000
                       102B7018 00000020 102B8A28 00000014
     +001A60 102B8A78  00000000 00000000 0000001D 00000000
                       00000000 00000000 00000000 00000000
```

Since B's anchor address was at 102B8A28, B's anchor is at 102B8A80, and B has, as expected, the hex value 0000001D or decimal 29.

Under NORENT NOWRITABLE(PRV)

The MAP listing when compiled with these options would look like this:

```
* * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *
 IDENTIFIER DEFINITION ATTRIBUTES

 ***TEST3     1-0:4  Class = ext def,  Location = CSECT ***TEST3
 ***TEST4     1-0:5  Class = ext def,  Location = CSECT ***TEST4
 _PRV_OFFSETS 1-0:1  Class = static,   Location = 8 : 0x8 + CSECT ***TEST2
```

The key here is the last line in this output: _PRV_OFFSETS is a static table that holds the offset into the PRV table for each CONTROLLED variable. This static table is generated only if the MAP option is specified.

To interpret this table, the compiler will also produce, immediately after the block names table, another, usually small, listing, which for our program would look like:

```
        PRV Offsets

   Number    Offset Name
        1        8 A
        1        C B
```

This table lists the hex offset within the runtime _PRV_OFFSETS table for each of the named CONTROLLED variables. The block number (in the first column) can be used to distinguish variables with the same name but declared in different blocks.

Since the _PRV_OFFSETS table is in static storage (at hex offset 8) and since PLIDUMP was called with the B option, it will appear in the dump output, which would look like this:

```
Static for procedure TEST    Timestamp: . . .
 +000000 10908EC8  02020240 00000005 6DD7D9E5 6DD6C6C6
                   00000000 00000004 D00000A0 00100000
 +000020 10908EE8  6E3BFFE0 00000000 00000000 00000000
                   00000000 90010000 00000000 00000000
```

So, the offset of A in the PRV table is 0, and the offset of B in the PRV table is 4. Note also the eyecatcher ″_PRV_OFF″ that occupies the first 8 bytes of the _PRV_OFFSETs table.

The PRV table is always located at offset 4 within the CAA which, since PLIDUMP was called with the H option, will be in the dump output and which looks like:

```
Control Blocks Associated with the Thread:
  CAA: 0A7107D0
   +000000 0A7107D0   00000800 0ADB7DE0 0AD97018 0ADB7018
                      00000000 00000000 00000000 00000000
```

So, the address of the PRV table is 0ADB7DE0, and it will also be in the dump
output amongst the HEAP storage:

```
 Enclave Storage:
    Initial (User) Heap
      +000000 102B7018   C8C1D5C3 0FC0F990 0FC0F990 00000000
       . . .
      +000DC0 0ADB7DD8   00000000 00000000 0ADB8A38 0ADB8A58
                         0ADB7018 00000488 00000000 00000000
```

So, the PRV table contains 0ADB8A38 0ADB8A58 etc, and since, as derived from
the _PRV_OFFSETS table, A's offset into the PRV table is 0 and B's offset is 4, these
are also the addresses of A and B respectively.

These addresses will also appear in the HEAP storage in the dump:

```
 Enclave Storage:
    Initial (User) Heap
      +000000 102B7018   C8C1D5C3 0FC0F990 0FC0F990 00000000
       . . .
      +001A00 0ADB8A18   00000000 00000000 0ADB7018 00000020
                         00000000 00000014 0A7107D4 00000000
      +001A20 0ADB8A38   00000000 00000000 0ADB7018 00000020
                         00000004 00000014 0A7107D4 00000000
      +001A40 0ADB8A58   0000001D 00000000 00000000 00000000
                         00000000 00000000 00000000 00000000
```

So, since A's address is 0ADB8A38, its hex value is, as expected, 00000000, and
since B's address is 0ADB8A58, its hex value is, also as expected, 0000001D.

Saved Compilation Data

During a compilation, the compiler saves various information about the
compilation in the load module. This information can be very useful in debugging
and in future migration efforts. This chapter describes the information saved.

Timestamp

The compiler saves in every load module a ″timestamp″, which is a 20-byte
character string of the form ″YYYYMMDDHHMISSVNRNML″ which records the
date and time of the compilation as well as the version of the compiler that
produced this string. The elements of the string have the following meanings:

YYYY
    the year of the compilation

MM
    the month of the compilation

DD
    the day of the compilation

HH
    the hour of the compilation

MI
    the minute of the compilation

SS the second of the compilation

VN
>   the version number of the compiler

RN
>   the release number of the compiler

ML
>   the maintenance level of the compiler

The timestamp can be located from the PPA2: at offset 12 in the PPA2 is a four-byte integer giving the offset (possibly negative) to the timestamp from the address of the PPA2.

The PPA2, in turn, can be located from the PPA1: at offset 4 in the PPA1 is a four-byte integer giving the offset (possibly negative) to the PPA2 from the entry point address corresponding to that PPA1.

The PPA1, in turn, can be located from the entry point address for a block: at offset 12 from the entry point address is a four-byte integer giving the offset (possibly negative) to the PPA1 from the entry point address.

Saved options string

The compiler also stores in the load module a 32-byte ″string″ that records the compiler options used in building the load module.

A PL/I declare for the saved options string is given in Figure 95. For most of the fields in the structure, the meaning of the field is obvious given its name, but a few of the fields need some explanation:

*   sos_words holds the number of the bytes in the structure divided by 4
*   sos_version is the version number for this structure. It is not a compiler version number.
*   the size of the structure and what fields have been set depends on the version number.

```
dcl
 1 sos   based
   ,2 sos_words              fixed bin(8) unsigned
   ,2 sos_version            fixed bin(8) unsigned
   ,2 sos_arch               fixed bin(8) unsigned
   ,2 sos_tune               fixed bin(8) unsigned
   ,2 sos_currency           char(1)
   ,2 sos_optlevel           bit(4)  /* set with version >= 2 */
   ,2 sos_scheduler          bit(1)  /* set with version >= 5 */
   ,2 sos_nowritable_prv     bit(1)  /* set with version >= 4 */
   ,2 sos_noblockedio        bit(1)  /* set with version >= 3 */
   ,2 sos_optimize           bit(1)
   ,2 sos_window             fixed bin(15)
   ,2 sos_codepage           fixed bin(31)
   ,2 sos_limits_intname     fixed bin(8) unsigned
   ,2 sos_limits_extname     fixed bin(8) unsigned
   ,2 sos_limits_fixbinp1    fixed bin(8) unsigned
   ,2 sos_limits_fixbinp2    fixed bin(8) unsigned
   ,2 sos_limits_fixdecp1    fixed bin(8) unsigned
   ,2 sos_limits_fixdecp2            /* set with version >= 4 */
                             fixed bin(8) unsigned
```

Figure 95. Declare for the saved options string (Part 1 of 3)

```
                 ,2 sos_flags1
                  ,3 sos_check_stg     bit(1)
                  ,3 sos_compact       bit(1)
                  ,3 sos_csect         bit(1)
                  ,3 sos_dbcs          bit(1)
                  ,3 sos_display_wto   bit(1)
                  ,3 sos_extrn_full    bit(1)
                  ,3 sos_graphic       bit(1)
                  ,3 sos_check_conform bit(1)  /* set with version >= 6 */
                 ,2 sos_flags2
                  ,3 sos_interrupt     bit(1)
                  ,3 sos_reduce        bit(1)
                  ,3 sos_norent        bit(1)
                  ,3 sos_respect_date  bit(1)
                  ,3 sos_rules_ans     bit(1)
                  ,3 sos_stdsys        bit(1)
                  ,3 sos_nowritable    bit(1)
                  ,3 sos_wchar_big     bit(1)
                 ,2 sos_flags3
                  ,3 sos_cmpat         bit(4)
                  ,3 sos_system        bit(4)
                 ,2 sos_flags4
                  ,3 sos_dllinit       bit(1)
                  ,3 sos_xinfo_def     bit(1)
                  ,3 sos_xinfo_xml     bit(1)
                  ,3 sos_static_full   bit(1)
                  ,3 sos_backreg_5     bit(1)
                  ,3 sos_noresexp      bit(1)  /* set with version >= 2 */
                  ,3 sos_bifprec       bit(2)  /* set with version >= 2 */
                                               /* 01 => bifprec(15)      */
                                               /* 10 => bifprec(31)      */
                 ,2 sos_test
                  ,3 sos_test_hooks    bit(4)
                  ,3 sos_test_sym      bit(1)
                  ,3 sos_test_nohook   bit(1)  /* set with version >= 5 */
                  ,3 sos_test_separate bit(1)  /* set with version >= 7 */
                  ,3 sos_Static_Length bit(1)  /* set with version >= 2 */
                 ,2 sos_float
                  ,3 sos_afp           bit(1)
                  ,3 sos_dft_nobin1arg bit(1)  /* set with version >>= 7 */
                  ,3 sos_dec_forcedsign bit(1)  /* set with version >= 6 */
                  ,3 sos_dec_nofoflonasgn bit(1)  /* set with version >= 6 */
                  ,3 sos_prectype      bit(2)  /* set with version >= 5 */
                  ,3 sos_floatinmath   bit(2)  /* set with version >= 2 */
                 ,2 sos_usage
                  ,3 sos_ans_round     bit(1)
                  ,3 sos_ans_unspec    bit(1)
                  ,3 sos_common        bit(1)  /* set with version >= 6 */
                  ,3 sos_initauto      bit(1)  /* set with version >= 5 */
                  ,3 sos_initbased     bit(1)  /* set with version >= 5 */
                  ,3 sos_initctl       bit(1)  /* set with version >= 5 */
                  ,3 sos_initstatic    bit(1)  /* set with version >= 5 */
                  ,3 sos_stringofg_is_c bit(1)  /* set with version >= 5 */
```

Figure 95. Declare for the saved options string (Part 2 of 3)

```
      ,2 sos_default
        ,3 sos_ans            bit(1)
        ,3 sos_asgn           bit(1)
        ,3 sos_byaddr         bit(1)
        ,3 sos_conn           bit(1)
        ,3 sos_descriptor     bit(1)
        ,3 sos_ebcdic         bit(1)
        ,3 sos_nonnative      bit(1)
        ,3 sos_nonnativeaddr  bit(1)
        ,3 sos_inline         bit(1)
        ,3 sos_reorder        bit(1)
        ,3 sos_evendec        bit(1)
        ,3 sos_null370        bit(1)
        ,3 sos_recursive      bit(1)
        ,3 sos_desclctr       bit(1)
        ,3 sos_ret_byaddr     bit(1)
        ,3 sos_initfill       bit(1)
        ,3 sos_initfill_char  char(1)
        ,3 sos_short_ieee     bit(1)
        ,3 sos_dummy_unal     bit(1)
        ,3 sos_retcode        bit(1)
        ,3 sos_unaligned      bit(1)
        ,3 sos_ordinal_max    bit(1)
        ,3 sos_overlap        bit(1)
        ,3 sos_hex            bit(1)
        ,3 sos_e_hex          bit(1)
        ,3 sos_linkage        fixed bin(8) unsigned
      ,2 sos_prefix
        ,3 sos_size           bit(1)
        ,3 sos_stringrange    bit(1)
        ,3 sos_stringsize     bit(1)
        ,3 sos_subrg          bit(1)
        ,3 sos_fofl           bit(1)
        ,3 sos_ofl            bit(1)
        ,3 sos_invalidop      bit(1)
        ,3 sos_ufl            bit(1)
        ,3 sos_zdiv           bit(1)
        ,3 sos_conv           bit(1)
        ,3 *                  bit(1)
        ,3 sos_dfp            bit(1)  /* set with version >= 9 */
        ,3 sos_nosepname      bit(1)  /* set with version >= 8 */
        ,3 sos_csectcut       bit(3)  /* set with version >= 5 */
      ,2 sos_extension01
        ,3 *                  bit(16)
        ,3 sos_cuname_offset  fixed bin(16) unsigned
```

Figure 95. Declare for the saved options string (Part 3 of 3)

The possible values for the sos_cmpat field are given by these declares:

```
 dcl sos_cmpat_le           bit(4) value('0000'b);
 dcl sos_cmpat_v1           bit(4) value('0001'b);
 dcl sos_cmpat_v2           bit(4) value('0010'b);
```

The possible values for the sos_system field are given by these declares:

```
dcl sos_system_mvs          bit(4) value('0001'b);
  dcl sos_system_tso        bit(4) value('0010'b);
  dcl sos_system_cics       bit(4) value('0011'b);
  dcl sos_system_ims        bit(4) value('0100'b);
  dcl sos_system_os         bit(4) value('0101'b);
```

The possible values for the sos_test_hooks field are given by these declares:

```
 dcl sos_test_hooks_none      bit(4) value('0000'b);
   dcl sos_test_hooks_block    bit(4) value('0001'b);
   dcl sos_test_hooks_stmt     bit(4) value('0011'b);
   dcl sos_test_hooks_path     bit(4) value('0101'b);
   dcl sos_test_hooks_all      bit(4) value('0111'b);
```

The possible values for the sos_linkage field are given by these declares:

```
   dcl sos_linkage_optlink     fixed bin(8) unsigned value(1);
   dcl sos_linkage_system      fixed bin(8) unsigned value(2);
```

The possible values for the sos_bifprec field are given by these declares:

```
 dcl sos_bifprec_15           bit(2) value('01'b);
   dcl sos_bifprec_31         bit(2) value('10'b);
```

The possible values for the sos_floatinmath field are given by these declares:

```
dcl sos_floatinmath_asis     bit(2) value('00'b);
   dcl sos_floatinmath_long   bit(2) value('10'b);
   dcl sos_floatinmath_extndd bit(2) value('11'b);
```

The saved options string is located after the timestamp in one of two ways:

1. if the service option has been specified, the string specified in the service option follows immediately after the timestamp as a character varying string. Then the saved options string follows after the service string as a second character varying string.

2. if the service option has not been specified, the saved options string follows immediately after the timestamp as a character varying string.

The length of the varying string that holds the saved options string may be longer than the size of the saved options string itself.

The presence (or absence) of the service string is indicated in the PPA2 by the flag byte at decimal offset 20 in the PPA2: if the result of anding this byte with '20'bx is not zero, then the service string is present.

In some earlier releases of the PL/I compiler, the compiler did not place a saved options string in the load module. The presence (or absence) of the saved options string is indicated in the PPA2 by the flag byte at decimal offset 20 in the PPA2: if the result of anding this byte with '02'bx is not zero, then the saved options string is present.

Chapter 17. Interrupts and attention processing

To enable a PL/I program to recognize attention interrupts, two operations must be possible:

- You must be able to create an interrupt. This is done in different ways depending upon both the terminal you use and the operating system.
- Your program must be prepared to respond to the interrupt. You can write an ON ATTENTION statement in your program so that the program receives control when the ATTENTION condition is raised.

> **Note:** If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following:
> - The INTERRUPT option (supported only in TSO)
> - A TEST option other than NOTEST or TEST(NONE,NOSYM).
>   Compiling this way causes INTERRUPT(ON) to be in effect, unless you explicitly specify INTERRUPT(OFF) in PLIXOPT.

You can find the procedure used to create an interrupt in the IBM instruction manual for the operating system and terminal that you are using.

There is a difference between the interrupt (the operating system recognized your request) and the raising of the ATTENTION condition.

An *interrupt* is your request that the operating system notify the running program. If a PL/I program was compiled with the INTERRUPT compile-time option, instructions are included that test an internal interrupt switch at discrete points in the program. The internal interrupt switch can be set if any program in the load module was compiled with the INTERRUPT compile-time option.

The internal switch is set when the operating system recognizes that an interrupt request was made. The execution of the special testing instructions (polling) raises the ATTENTION condition. If a debugging tool hook (or a CALL PLITEST) is encountered before the polling occurs, the debugging tool can be given control before the ATTENTION condition processing starts.

Polling ensures that the ATTENTION condition is raised between PL/I statements, rather than within the statements.

Figure 96 on page 380 shows a skeleton program, an ATTENTION ON-unit, and several situations where polling instructions will be generated. In the program polling will occur at:

- LABEL1
- Each iteration of the DO
- The ELSE PUT SKIP ... statement
- Block END statements

```
%PROCESS INTERRUPT;
    .
    .
    .
 ON ATTENTION
   BEGIN;
     DCL X FIXED BINARY(15);
     PUT SKIP LIST ('Enter 1 to terminate, 0 to continue.');
     GET SKIP LIST (X);
     IF X = 1 THEN
       STOP;
     ELSE
       PUT SKIP LIST ('Attention was ignored');
   END;
    .
    .
    .
  LABEL1:
   IF EMPNO ...
    .
    .
    .
  DO I = 1 TO 10;
    .
    .
    .
  END;
    .
    .
    .
```

Figure 96. Using an ATTENTION ON-unit

Using ATTENTION ON-units

You can use processing within the ATTENTION ON-unit to terminate potentially endless looping in a program.

Control is given to an ATTENTION ON-unit when polling instructions recognize that an interrupt has occurred. Normal return from the ON-unit is to the statement following the polling code.

Interaction with a debugging tool

If the program has the TEST(ALL) or TEST(ERROR) run-time option in effect, an interrupt causes the debugging tool to receive control the next time a hook is encountered. This might be before the program's polling code recognizes that the interrupt occurred.

Later, when the ATTENTION condition is raised, the debugging tool receives control again for condition processing.

Chapter 18. Using the Checkpoint/Restart facility

This chapter describes the PL/I Checkpoint/Restart feature which provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment.

At points specified in the program, information about the current status of the program is written as a record on a data set. If the program terminates due to a system failure, you can use this information to restart the program close to the point where the failure occurred, avoiding the need to rerun the program completely.

This restart can be either automatic or deferred. An automatic restart is one that takes place immediately (provided the operator authorizes it when requested by a system message). A deferred restart is one that is performed later as a new job.

You can request an automatic restart from within your program without a system failure having occurred.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system. This facility is described in the books listed in "Bibliography" on page 447.

To use checkpoint/restart you must do the following:
- Request, at suitable points in your program, that a checkpoint record is written. This is done with the built-in subroutine PLICKPT.
- Provide a data set on which the checkpoint record can be written.
- Also, to ensure the desired restart activity, you might need to specify the RD parameter in the EXEC or JOB statement (see *z/OS JCL Reference*).

Note: You should be aware of the restrictions affecting data sets used by your program. These are detailed in the "Bibliography" on page 447.

Requesting a checkpoint record

Each time you want a checkpoint record to be written, you must invoke, from your PL/I program, the built-in subroutine PLICKPT.

```
►►──CALL──PLICKPT─────────────────────────────────────────►◄
                  └─(─ddname──────────────────────────)─┘
                         └─,──check-id───────────────┘
                                     └─,──org──────┘
                                            └─,──code─┘
```

The four arguments are all optional. If you do not use an argument, you need not specify it unless you specify another argument that follows it in the given order. In this case, you must specify the unused argument as a null string ('). The following paragraphs describe the arguments.

ddname
    is a character string constant or variable specifying the name of the DD

statement defining the data set that is to be used for checkpoint records. If you omit this argument, the system will use the default ddname SYSCHK.

check-id
> is a character string constant or variable specifying the name that you want to assign to the checkpoint record so that you can identify it later. If you omit this argument, the system will supply a unique identification and print it at the operator's console.

org
> is a character string constant or variable with the attributes CHARACTER(2) whose value indicates, in operating system terms, the organization of the checkpoint data set. PS indicates sequential (that is, CONSECUTIVE) organization; PO represents partitioned organization. If you omit this argument, PS is assumed.

code
> is a variable with the attributes FIXED BINARY (31), which can receive a return code from PLICKPT. The return code has the following values:

0	A checkpoint has been successfully taken.
4	A restart has been successfully made.
8	A checkpoint has not been taken. The PLICKPT statement should be checked.
12	A checkpoint has not been taken. Check for a missing DD statement, a hardware error, or insufficient space in the data set. A checkpoint will fail if taken while a DISPLAY statement with the REPLY option is still incomplete.
16	A checkpoint has been taken, but ENQ macro calls are outstanding and will not be restored on restart. This situation will not normally arise for a PL/I program.

Defining the checkpoint data set

You must include a DD statement in the job control procedure to define the data set in which the checkpoint records are to be placed. This data set can have either CONSECUTIVE or partitioned organization. You can use any valid ddname. If you use the ddname SYSCHK, you do not need to specify the ddname when invoking PLICKPT.

You must specify a data set name only if you want to keep the data set for a deferred restart. The I/O device can be any direct access device.

To obtain only the last checkpoint record, then specify status as NEW (or OLD if the data set already exists). This will cause each checkpoint record to overwrite the previous one.

To retain more than one checkpoint record, specify status as MOD. This will cause each checkpoint record to be added after the previous one.

If the checkpoint data set is a library, "check-id" is used as the member-name. Thus a checkpoint will delete any previously taken checkpoint with the same name.

For direct access storage, you should allocate enough primary space to store as many checkpoint records as you will retain. You can specify an incremental space

allocation, but it will not be used. A checkpoint record is approximately 5000 bytes longer than the area of main storage allocated to the step.

No DCB information is required, but you can include any of the following, where applicable:

```
OPTCD=W, OPTCD=C, RECFM=UT
```

These subparameters are described in the *z/OS JCL User's Guide*.

Requesting a restart

A restart can be automatic or deferred. You can make automatic restarts after a system failure or from within the program itself. The system operator must authorize all automatic restarts when requested by the system.

Automatic restart after a system failure

If a system failure occurs after a checkpoint has been taken, the automatic restart will occur at the last checkpoint if you have specified RD=R (or omitted the RD parameter) in the EXEC or JOB statement.

If a system failure occurs before any checkpoint has been taken, an automatic restart, from the beginning of the job step, can still occur if you have specified RD=R in the EXEC or JOB statement.

After a system failure occurs, you can still force automatic restart from the beginning of the job step by specifying RD=RNC in the EXEC or JOB statement. By specifying RD=RNC, you are requesting an automatic step restart without checkpoint processing if another system failure occurs.

Automatic restart within a program
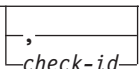
You can request a restart at any point in your program. The rules for the restart are the same as for a restart after a system failure. To request the restart, you must execute the statement:

```
CALL PLIREST;
```

To effect the restart, the compiler terminates the program abnormally, with a system completion code of 4092. Therefore, to use this facility, the system completion code 4092 must not have been deleted from the table of eligible codes at system generation.

Getting a deferred restart

To ensure that automatic restart activity is canceled, but that the checkpoints are still available for a deferred restart, specify RD=NR in the EXEC or JOB statement when the program is first executed.

```
►►──RESTART──=──(──stepname─┬──────────┬──)──────────────────────►◄
                           │    ,     │
                           └─check-id─┘
```

If you subsequently require a deferred restart, you must submit the program as a new job, with the RESTART parameter in the JOB statement. Use the RESTART parameter to specify the job step at which the restart is to be made and, if you want to restart at a checkpoint, the name of the checkpoint record.

For a restart from a checkpoint, you must also provide a DD statement that defines the data set containing the checkpoint record. The DD statement must be named SYSCHK. The DD statement must occur immediately before the EXEC statement for the job step.

Modifying checkpoint/restart activity

You can cancel automatic restart activity from any checkpoints taken in your program by executing the statement:

```
CALL PLICANC;
```

However, if you specified RD=R or RD=RNC in the JOB or EXEC statement, automatic restart can still take place from the beginning of the job step.

Also, any checkpoints already taken are still available for a deferred restart.

You can cancel any automatic restart and the taking of checkpoints, even if they were requested in your program, by specifying RD=NC in the JOB or EXEC statement.

Chapter 19. Using user exits

PL/I provides a number of user exits that allow you to customize the PL/I product to suit your needs. The PL/I products supply default exits and the associated source files.

If you want the exits to perform functions that are different from those supplied by the default exits, we recommend that you modify the supplied source files as appropriate.

At times, it is useful to be able to tailor the compiler to meet the needs of your organization. For example, you might want to suppress certain messages or alter the severity of others. You might want to perform a specific function with each compilation, such as logging statistical information about the compilation into a file. A compiler user exit handles this type of function.

With PL/I, you can write your own user exit or use the exit provided with the product, either 'as is' or modified, depending on what you want to do with it. The user exit source code provided with the product can be seen in Figure 16 on page 149.

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 97, 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.
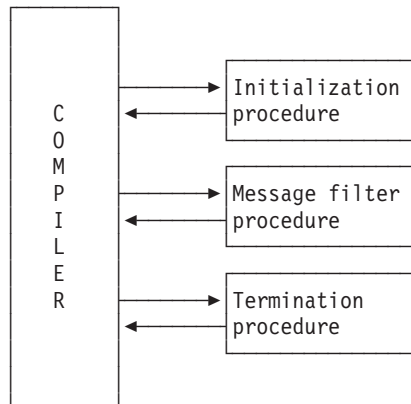
Figure 97. PL/I compiler user exit procedures

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 387.
- 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 388, "Writing the message filtering procedure" on page 389, and "Writing the termination procedure" on page 390.

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 27.

The EXIT compile-time option allows you to specify a user-option-string which specifies the DDname for the user exit input file. If you do not specify a string, SYSUEXIT is used as the DDname for the user exit input file.

The user-option-string is passed to the user exit functions in the global control block which is discussed in "Structure of global control blocks" on page 387. Please refer to the field "Uex_UIB_User_char_str" in the section "Structure of global control blocks" on page 387 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.

Customizing the compiler user exit

As was mentioned earlier, you can write your own compiler user exit or simply use the one shipped with the compiler. In either case, the name of the fetchable file for the compiler user exit must be IBMUEXIT.

This section describes how to:
- Modify the user exit input file for customized message filtering

- Create your own compiler user exit

Modifying SYSUEXIT

Rather than spending the time to write a completely new compiler user exit, you can simplify modify the user exit input file.

Edit the file to indicate which message numbers you want to suppress, and which message number severity levels you would like changed. A sample file is shown in Figure 98.

```
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-------------------------------
  'IBM'     1042      -1          1        String spans multiple lines
  'IBM'     1044      -1          1        FIXED BIN 7 mapped to 1 byte
  'IBM'     1047       8          0        Order inhibits optimization
  'IBM'     1052      -1          1        Nodescriptor with * extent arg
  'IBM'     1059       0          0        Select without OTHERWISE
  'IBM'     1169       0          1        Precision of result determined
```

Figure 98. Example of an user exit input file

The first two lines are header lines and are ignored by IBMUEXIT. The remaining lines contain input separated by a variable number of blanks.

Each column of the file is relevant to the compiler user exit:
- The first column should contain the letters 'IBM' in single quotes for all compiler messages to which you want the exit to apply. For messages from the SQL side of the SQL preprocessor, it should contain the SQL message prefix 'SQL' (again in single quotes).
- 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 (see the source in Figure 16.) as a model. As you write the exit, make sure it covers the areas of initialization, message filtering, and termination.

As noted in that section, the compiler user exit must be compiled with the RENT option and linked as a DLL.

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_ISA) *  /
```

The global control block syntax for the initialization procedure is discussed in the section "Structure of global control blocks" on page 387.

Upon completion of the initialization procedure, you should set the return/reason codes to the following:

0/0
> Continue compilation

4/n
> Reserved for future use

8/n
> Reserved for future use

12/n
> Reserved for future use

16/n
> Abort compilation

Writing the message filtering procedure

The message filtering procedure permits you to either suppress messages or alter the severity of messages. You can increase the severity of any of the messages but you can decrease the severity only of **ERROR** (severity code 8) or **WARNING** (severity code 4) messages.

The procedure-specific control block contains information about the messages. It is used to pass information back to the compiler indicating how a particular message should be handled.

The following is an example of a procedure-specific message filter control block:

```
Dcl 1 Uex_MFX native based( null() ),
    2 Uex_MFX_Length   fixed bin(31),

    2 Uex_MFX_Facility_Id  char(3),        /* of component writing
                                              message          */
    2 *                 char(1),
    2 Uex_MFX_Message_no   fixed bin(31),
    2 Uex_MFX_Severity     fixed bin(15),
    2 Uex_MFX_New_Severity fixed bin(15),  /* set by exit proc   */
    2 Uex_MFX_Inserts              fixed bin(15),
    2 Uex_MFX_Inserts_Data( 6 refer(Uex_MFX_Inserts) ),
      3 Uex_MFX_Ins_Type             fixed bin(7),
      3 Uex_MFX_Ins_Type_Data union unaligned,
        4 *                          char(8),
        4 Uex_MFX_Ins_Bin            fixed bin(31),
        4 Uex_MFX_Ins_Str,
          5 Uex_MFX_Ins_Str_Len      fixed bin(15),
          5 Uex_MFX_Ins_Str_Addr     pointer,
        4 Uex_MFX_Ins_Series,
          5 Uex_MFX_Ins_Series_Sep   char(1),
          5 Uex_MFX_Ins_Series_Addr  pointer;
```

Data Entry Fields
- **Uex_MFX_Length**: Contains the length of the control block in bytes. The value is storage (Uex_MFX).
- **Uex_MFX_Facility_Id**: Contains the ID of the facility; for the compiler, the ID is IBM. For the SQL side of the SQL preprocessor, the id is SQL. The compiler sets this value.

- **Uex_MFX_Message_no**: Contains the message number that the compiler is going to generate. The compiler sets this value.
- **Uex_MFX_Severity**: Contains the severity level of the message; it can be from one to fifteen characters in length. The compiler sets this value.
- **Uex_MFX_New_Severity**: Contains the new severity level of the message; it can be from one to fifteen characters in length. The user sets this value.
- **Uex_MFX_Inserts**: Contains the number of inserts for the message; it can range from zero to six. The compiler sets this value.
- **Uex_MFX_Inserts_Data**: Contains fields to describe each of the inserts. The compiler sets these values.
- **Uex_MFX_Ins_Type**: Contains the type of the insert. The possible insert types are:
  - **Uex_Ins_Type_Xb31**: Used for an integer type and has the value 1.
  - **Uex_Ins_Type_Char**: Used for an integer type and has the value 2.
  - **Uex_Ins_Type_Series**: Used for an integer type and has the value 3.

  The compiler sets this value.
- **Uex_MFX_Ins_Bin**: Contains the integer value for an insert that has integer type. The compiler sets this value.
- **Uex_MFX_Ins_Str_Len**: Contains the length (in bytes) for an insert that has character type. The compiler sets this value.
- **Uex_MFX_Ins_Str_Addr**: Contains the address of the character string for an insert that has character type. The compiler sets this value.
- **Uex_MFX_Ins_Series_Sep**: Contains the character that should be inserted between each element for an insert that has series type. Typically, this is a blank, period or comma. The compiler sets this value.
- **Uex_MFX_Ins_Series_Addr**: Contains the address of the series of varying character strings for an insert that has series type. The address points to a FIXED BIN(31) field holding the number of strings to concatenate followed by the adrresses of those strings. The compiler sets this value.

Upon completion of the message filtering procedure, set the return/reason codes to one of the following:

0/0
Continue compilation, output message

0/1
Continue compilation, do not output message

4/n
Reserved for future use

8/n
Reserved for future use

16/n
Abort compilation

Writing the termination procedure

You should use the termination procedure to perform any cleanup required, such as closing files. You might also want to write out final statistical reports based on information collected during the error message filter procedures and the initialization procedures.

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 387. 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

Chapter 20. PL/I descriptors under CMPAT(LE)

This chapter describes PL/I parameter passing conventions between PL/I routines at run time when you compile with the CMPAT(LE) option. For additional information about Language Environment run-time environment considerations, other than descriptors, see *z/OS Language Environment Programming Guide.* This includes run-time environment conventions and assembler macros supporting these conventions.

Note: The descriptors and argument passing mechanisms used under the compiler options CMPAT(V1) and CMPAT(V2) are the same as those generated under OS PL/I and are not discussed here.

Passing an argument

When a string, an array, or a structure is passed as an argument, the compiler passes a descriptor for that argument unless the called routine is declared with OPTIONS(NODESCRIPTOR). There are two methods for passing such descriptors:
- By descriptor list
- By descriptor locator

The following key features should be noted about each of these two methods:
- **When arguments are passed with a descriptor list**
  - The number of arguments passed is one greater than the number of arguments specified if any of the arguments needs a descriptor.
  - An argument passed with a descriptor can be received as a pointer passed by value (BYVALUE).
  - The compiler uses this method when the DEFAULT(DESCLIST) compiler option is in effect.
- **When arguments are passed by descriptor locator**
  - The number of arguments passed always matches the number of arguments specified.
  - An argument passed with a descriptor can be received as a pointer passed by address (BYADDR).
  - The compiler uses this method when the DEFAULT(DESCLOCATOR) compiler option is in effect.

Argument passing by descriptor list

When arguments and their descriptors are passed with a descriptor list, an extra argument is passed whenever at least one argument needs a descriptor. This extra argument is a pointer to a list of pointers. The number of entries in this list equals the number of arguments passed. For arguments that don't require a descriptor, the corresponding pointer in the descriptor list is set to SYSNULL. For arguments that do require a descriptor, the corresponding pointer in the descriptor list is set to the address of that argument's descriptor.

So, for example, suppose the routine `sample` is declared as

```
declare sample entry( fixed bin(31), varying char(*) )
             options( byaddr descriptor );
```

Then, if `sample` is called as in the following statement:

```
call sample( 1, 'test' );
```

The following three arguments are passed to the routine:
- Address of a fixed bin(31) temporary with the value 1
- Address of a varying char(4) temporary with the value `test`
- Address of a descriptor list consisting of the following:
  - SYSNULL()
  - Address of the descriptor for a varying char(4) string

Argument passing by descriptor-locator

When arguments and their descriptors are passed by descriptor-locator, whenever an argument requires a descriptor, the address of a locator/descriptor for it is passed instead.

The locator/descriptor is a pair of pointers. The first pointer is the address of the data; the second pointer is the address of the descriptor.

So, for example, suppose the routine `sample` is declared again as

```
declare sample entry( fixed bin(31), varying char(*) )
               options( byaddr descriptor );
```

Then, if `sample` is called as in the following statement

```
call sample( 1, 'test' );
```

The following two arguments are passed to the routine:
- Address of a fixed bin(31) temporary with the value 1
- Address of a descriptor-locator consisting of the following:
  - Address of a varying char(4) temporary with the value `test`
  - Address of the descriptor for a varying char(4) string

Descriptor header

Every descriptor starts with a 4-byte field. The first byte specifies the descriptor type (scalar, array, structure or union). The remaining three bytes are zero unless they are set by the particular descriptor type.

The declare for a descriptor header is:

```
declare
  1 dsc_Header based( sysnull() ),
    2 dsc_Type        fixed bin(8) unsigned,
    2 dsc_Datatype    fixed bin(8) unsigned,
    2 *               fixed bin(8) unsigned,
    2 *               fixed bin(8) unsigned;
```

The possible values for the dsc_Type field are:

```
declare
  dsc_Type_Unset            fixed bin(8) value(0),
  dsc_Type_Element          fixed bin(8) value(2),
  dsc_Type_Array            fixed bin(8) value(3),
  dsc_Type_Structure        fixed bin(8) value(4),
  dsc_Type_Union            fixed bin(8) value(4);
```

String descriptors

In a string descriptor, the second byte of the header indicates the string type (bit, character or graphic as well as nonvarying, varying or varyingz).

In a string descriptor for a nonvarying bit string, the third byte of the header gives the bit offset.

In a string descriptor for a varying string, the fourth byte has a bit indicating if the string length is held in nonnative format.

In a string descriptor for a character string, the fourth byte also has a bit indicating if the string data is in EBCDIC.

The declare for a string descriptor is:

```
declare
  1 dsc_String  based( sysnull() ),
    2 dsc_String_Header,
      3 *                  fixed bin(8) unsigned,
      3 dsc_String_Type    fixed bin(8) unsigned,
      3 dsc_String_BitOfs fixed bin(8) unsigned,
      3 *,
        4 dsc_String_Has_Nonnative_Len     bit(1),
        4 dsc_String_Is_Ebcdic             bit(1),
        4 dsc_String_Has_Nonnative_Data    bit(1),
        4 *                                bit(5),
    2 dsc_String_Length   fixed bin(31); /* max length of string */
```

The possible values for the dsc_String_Type field are:

```
declare
  dsc_String_Type_Unset              fixed bin(8) value(0),
  dsc_String_Type_Char_Nonvarying    fixed bin(8) value(2),
  dsc_String_Type_Char_Varyingz      fixed bin(8) value(3),
  dsc_String_Type_Char_Varying2      fixed bin(8) value(4),
  dsc_String_Type_Bit_Nonvarying     fixed bin(8) value(6),
  dsc_String_Type_Bit_Varying2       fixed bin(8) value(7),
  dsc_String_Type_Graphic_Nonvarying fixed bin(8) value(9),
  dsc_String_Type_Graphic_Varyingz   fixed bin(8) value(10),
  dsc_String_Type_Graphic_Varying2   fixed bin(8) value(11),
  dsc_String_Type_Widechar_Nonvarying fixed bin(8) value(13),
  dsc_String_Type_Widechar_Varyingz   fixed bin(8) value(14),
  dsc_String_Type_Widechar_Varying2   fixed bin(8) value(15);
```

Array descriptors

The declare for an array descriptor is:

```
declare
  1 dsc_Array  based( sysnull() ),
    2 dsc_Array_Header   like dsc_Header,
    2 dsc_Array_EltLen   fixed bin(31), /* Length of array element */
    2 dsc_Array_Rank     fixed bin(31), /* Count of dimensions     */
    2 dsc_Array_RVO      fixed bin(31), /* Relative virtual origin */
    2 dsc_Array_Data( 1: 1 refer(dsc_Array_Rank) ),
      3 dsc_Array_LBound fixed bin(31), /*   LBound               */
      3 dsc_Array_Extent fixed bin(31), /*   HBound - LBound + 1  */
      3 dsc_Array_Stride fixed bin(31); /*   Multiplier           */
```

Part 6. Appendixes

Appendix. SYSADATA message information

When you specify the MSG suboption of the XINFO compile-time option, the compiler generates a SYSADATA file that contains:

- Counter records
- Literal records
- File records
- Message records

You should note that the records in the file are not necessarily produced in the order listed above, for example, literal and file records might be interleaved. If you are writing code that reads a SYSADATA file, you should not rely on the order of the records in the file except for the following exceptions:

- Counter records are the first records in the file.
- Each literal record precedes any reference to the literal it defines.
- Each file record precedes any reference to the file it describes.

Understanding the SYSADATA file

The SYSADATA file is a sequential binary file. Under z/OS batch, the compiler writes the SYSADATA records to the file specified by the SYSADATA DD statement, and that file must not be a member of a PDS. On all other systems, the compiler writes to a file with the extension ″adt″.

Each record in the file contains a header. This 8-byte header has fields that are the same for all records in the file:

Compiler
A number representing the compiler that produced the data. For PL/I, the number is 40.

Edition number
The edition number of the compiler that produced the data. For this product, it is the number 2.

SYSADATA level
A number representing the level of SYSADATA that this file format represents. For this product, it is the number 4.

The header also has some fields that vary from record to record:

- Record type
- Whether or not the record is continued onto the next record

Possible record types are encoded as an ordinal value as shown in Figure 99 on page 400.

```
Define ordinal xin_Rect
 (Xin_Rect_Msg       value(50),   /*  Message record               */
  Xin_Rect_Fil       value(57),   /*  File record                  */
  Xin_Rect_Sum       value(61),   /*  Summary record               */
  Xin_Rect_Src       value(63),   /*  Source record                */
  Xin_Rect_Tok       value(64),   /*  Token record                 */
  Xin_Rect_Sym       value(66),   /*  Symbol record                */
  Xin_Rect_Lit       value(67),   /*  Literal record               */
  Xin_Rect_Syn       value(69),   /*  Syntax record                */
  Xin_Rect_Ord_Type  value(80),   /*  ordinal type record          */
  Xin_Rect_Ord_Elem  value(81),   /*  ordinal element record       */
  Xin_Rect_Ctr       value(82) )  /*  counter record               */
                     prec(15);
```

Figure 99. Record types encoded as an ordinal value

The declare for the header part of a record is shown in Figure 100.

```
Dcl
  1 Xin_Hdr Based( null() ),       /*  Header portion               */
                                   /*                               */
    2 Xin_Hdr_Prod                 /*  Language code                */
          fixed bin(8) unsigned, /*                               */
                                   /*                               */
    2 Xin_Hdr_Rect                 /*  Record type                  */
       unal ordinal xin_Rect,      /*                               */
                                   /*                               */
    2 Xin_Hdr_Level                /*  SYSADATA level               */
          fixed bin(8) unsigned, /*                               */
                                   /*                               */
    2 * union,                     /*                               */
      3 xin_Hdr_Flags    bit(8),   /*  flags                        */
      3 *,                         /*                               */
        4 *              bit(6),   /*  Reserved                     */
        4 Xin_Hdr_Little_Endian    /*  ints are little endian       */
                         bit(1),   /*                               */
        4 Xin_Hdr_Cont   bit(1),   /*  Record continued in next rec */
                                   /*                               */
    2 Xin_Hdr_Edition              /*  compiler "edition"           */
          fixed bin(8) unsigned, /*                               */
                                   /*                               */
    2 Xin_Hdr_Fill      bit(32), /*  reserved                     */
                                   /*                               */
    2 Xin_Hdr_Data_Len             /*  length of data part          */
          fixed bin(16) unsigned, /*                               */
                                   /*                               */
    2 Xin_Hdr_End       char(0); /*                               */
```

Figure 100. Declare for the header part of a record

Summary Record

The record with type Xin_Rect_Sum, the summary record, will be the first record
in the file and its declare is shown in Figure 101 on page 401

```
Dcl
  1 Xin_Sum     Based( null() ),   /* summary record              */
                                   /*                             */
    2 Xin_Sum_Hdr                  /* standard header             */
         like Xin_Hdr,             /*                             */
                                   /*                             */
    2 Xin_Sum_Max_Severity         /* max severity from compiler  */
         fixed bin(32) unsigned,   /*                             */
                                   /*                             */
    2 Xin_Sum_Left_Margin          /* left margin                 */
         fixed bin(16) unsigned,   /*                             */
                                   /*                             */
    2 Xin_Sum_Right_Margin         /* right margin                */
         fixed bin(16) unsigned,   /*                             */
                                   /*                             */
    2 xin_Sum_Rsrvd(15)            /* reserved                    */
         fixed bin(32) unsigned;   /*                             */
```

Figure 101. Declare for a summary record

Counter records

Each counter record specifies, for a subsequent record type, how many records of
that type the file contains and how many bytes those records occupy.

```
Dcl
  1 xin_Ctr     Based( null() ),   /* counter/size record         */
                                   /*                             */
    2 xin_Ctr_Hdr                  /* standard header             */
         like xin_Hdr,             /*                             */
                                   /*                             */
    2 xin_Ctr_Rect                 /* record type                 */
       unal ordinal xin_Rect,      /*                             */
                                   /*                             */
    2 *                            /*                             */
         fixed bin(16) unsigned,   /*                             */
                                   /*                             */
    2 xin_Ctr_Count                /* count of that record type   */
         fixed bin(31) unsigned,   /*                             */
                                   /*                             */
    2 xin_Ctr_Size                 /* size used                   */
         fixed bin(31) unsigned;   /*                             */
```

Figure 102. Declare for a counter record

Literal records

Each literal record assigns a number, called a literal index, that is used by later
records to refer to the characters named in this particular record.

```
Dcl
 1 xin_Lit    Based( null() ),   /* literal record            */
                                 /*                           */
    2 xin_Lit_Hdr                /* standard header           */
         like xin_Hdr,           /*                           */
                                 /*                           */
    2 xin_Lit_Inx                /* adata index for literal   */
         fixed bin(31) unsigned, /*                           */
                                 /*                           */
    2 xin_Lit_Len                /* length of literal         */
         fixed bin(31) unsigned, /*                           */
                                 /*                           */
    2 xin_Lit_Val     char(2000); /* literal value            */
```

Figure 103. Declare for a literal record

File records

Each file record assigns a number, called a file index, that is used by later records
to refer to the file described by this record. The described file may be the primary
PL/I source file or an INCLUDEd file. Each file record specifies a literal index for
the fully qualified name of the file.

For an INCLUDEd file, each file record also contains the file index and source line
number from whence the INCLUDE request came. (For primary source files, these
fields are zero.)

```
Dcl
 1 xin_Fil    Based( null() ),   /* file record               */
                                 /*                           */
    2 xin_Fil_Hdr                /* standard header           */
         like xin_Hdr,           /*                           */
                                 /*                           */
    2 xin_Fil_File_Id            /* file id from whence it    */
         fixed bin(31) unsigned, /*  was INCLUDEd             */
                                 /*                           */
    2 xin_Fil_Line_No            /* line no within that file  */
         fixed bin(31) unsigned, /*                           */
                                 /*                           */
    2 xin_Fil_Id                 /* id assigned to this file  */
         fixed bin(31) unsigned, /*                           */
                                 /*                           */
    2 xin_Fil_Name               /* literal index of the      */
         fixed bin(31) unsigned; /*  fully qualified file name */
```

Figure 104. Declare for a file record

Message records

Each message record describes a message issued during the compilation. Message
records are not generated for suppressed messages.

Each message record contains:
• The file index and source line number for the file and line to which the message
  is attributed. If the message pertains to the compilation as a whole, these fields
  are zero.

- The identifier (e.g. IBM1502) and severity associated with the message.
- The text of the message.

The declare for a message record is as follows:

```
Dcl
  1 xin_Msg     Based( null() ),   /* message record              */
                                   /*                             */
    2 xin_Msg_Hdr                  /* standard header             */
          like xin_Hdr,            /*                             */
                                   /*                             */
    2 xin_Msg_File_Id              /* file id                     */
          fixed bin(31) unsigned,  /*                             */
                                   /*                             */
    2 xin_Msg_Line_No              /* line no within file         */
          fixed bin(31) unsigned,  /*                             */
                                   /*                             */
    2 xin_Msg_Id                   /* identifier (i.e. IBM1502)   */
                      char(16),    /*                             */
                                   /*                             */
    2 xin_Msg_Severity             /* severity (0, 4, 8, 12 or 16) */
          fixed bin(15)   signed,  /*                             */
                                   /*                             */
    2 xin_Msg_Length               /* length of message           */
          fixed bin(16) unsigned,  /*                             */
                                   /*                             */
    2 Xin_Msg_Text                 /* actual message              */
        char( 100 refer(xin_Msg_Length) );
```

Figure 105. Declare for a message record

Understanding SYSADATA symbol information

When you specify the SYM suboption of the XINFO compile-time option, the compiler generates a SYSADATA file that contains the following information in addition to the records generated for the MSG suboption:
- Ordinal type records
- Ordinal element records
- Symbol records

Symbol records are not generated for built-in functions, generic variables or variables with non-constant extents.

Ordinal type records

Each ordinal type record assigns a number, called an ordinal type index, that is used by later records to refer to an ordinal type described by this record. The name of the type is indicated by a literal index. Each ordinal type record contains the file index and source line number for the file and line in which the ordinal type was declared.

Each ordinal type record contains:
- A count of the number of values defined by that type
- The precision associated with the type
- Bits indicating if it is signed or unsigned

```
declare                          /*                          */
  1 xin_Ord_Type   based( null() ), /*                       */
                                 /*                          */
    2 xin_Ord_Type_Hdr            /* standard header         */
        like xin_Hdr,            /*                          */
                                 /*                          */
    2 xin_Ord_Type_File_Id        /* file id                 */
        fixed bin(31) unsigned,  /*                          */
                                 /*                          */
    2 xin_Ord_Type_Line_No        /* line no within file     */
        fixed bin(31) unsigned,  /*                          */
                                 /*                          */
    2 xin_Ord_Type_Id             /* identifying number      */
                fixed bin(31),    /*                          */
                                 /*                          */
    2 xin_Ord_Type_Count          /* count of elements       */
                fixed bin(31),    /*                          */
                                 /*                          */
    2 xin_Ord_Type_Prec           /* precision for ordinal    */
        fixed bin(08) unsigned,  /*                          */
                                 /*                          */
    2 *,                          /*                          */
      3 xin_Ordinal_Type_Signed   /* signed attribute applies */
                    bit(1),       /*                          */
      3 xin_Ordinal_Type_Unsigned /* unsigned attribute applies */
                    bit(1),       /*                          */
      3 *                         /* unused                   */
                    bit(6),       /*                          */
                                 /*                          */
    2 *                           /* unused                   */
                    char(2),      /*                          */
                                 /*                          */
    2 xin_Ord_Type_Name           /* type name               */
                fixed bin(31);    /*                          */
```

Figure 106. Declare for an ordinal type record

Ordinal element records

Each ordinal type record is immediately followed by a series of records (as many as specified by the ordinal type count) that describes the values named by that ordinal.

Each ordinal element record assigns a number, called an ordinal element index, that is used by later records to refer to an ordinal element described by this record. The name of the element is indicated by a literal index. Each ordinal element record contains the file index and source line number for the file and line in which the ordinal element was declared.

Additionally, each ordinal element record contains
- The ordinal type index of the ordinal type to which it belongs
- The value for that element

```
declare                            /*                          */
  1 xin_Ord_Elem   based( null() ), /*                         */
                                   /*                          */
    2 xin_Ord_Elem_Hdr              /* standard header         */
         like xin_Hdr,             /*                          */
                                   /*                          */
    2 xin_Ord_Elem_File_Id          /* file id                 */
         fixed bin(31) unsigned,   /*                          */
                                   /*                          */
    2 xin_Ord_Elem_Line_No          /* line no within file     */
         fixed bin(31) unsigned,   /*                          */
                                   /*                          */
    2 xin_Ord_Elem_Id               /* identifying number      */
              fixed bin(31),       /*                          */
                                   /*                          */
    2 xin_Ord_Elem_Type_Id          /* id of ordinal type      */
              fixed bin(31),       /*                          */
                                   /*                          */
    2 xin_Ord_Elem_Value            /* ordinal value           */
              fixed bin(31),       /*                          */
                                   /*                          */
    2 xin_Ord_Elem_Name             /* ordinal name            */
              fixed bin(31);       /*                          */
```

Figure 107. Declare for an ordinal element record

Symbol records

Each symbol record assigns a number, called a symbol index, that is used by later records to refer to the symbol (for example, the name of a user variable or constant) described by this record. The name of the identifier is indicated by a literal index. Each symbol record contains the file index and source line number for the file and line in which the symbol was declared.

If the identifier is part of a structure or union, the symbol record contains a symbol index for each of the following:

- The first sibling, if any
- The parent, if any
- The first child, if any

Consider the following structure:

```
dcl
  1 a
    , 3 b      fixed bin
    , 3 c      fixed bin
    , 3 d
      , 5 e    fixed bin
      , 5 f    fixed bin
  ;
```

The symbol indices assigned to the elements of the preceding structure would be as follows:

```
symbol   index    sibling    parent    child
-----    -----    -------    ------    -----
  a        1         0          0        2
  b        2         3          1        0
  c        3         4          1        0
  d        4         0          1        5
  e        5         6          4        0
  f        6         0          4        0
```

Figure 108. Symbol indices assigned to the elements of a structure

Each symbol record also contains a series of bit(1) fields that indicate if various attributes apply to this variable.

Each symbol record also contains the following elements:

User-given structure level
> This is a user-given structure level for the identifier. For the element c of the structure above, the value is 3. For non-structure members, the value is set to 1.

Logican structure level
> The logical structure level for the identifier For the element c of the structure above, the value is 2. For non-structure members, the value is set to 1.

Dimensions

> The number of dimensions declared for the variable not counting any inherited dimensions.

> The number of dimensions for the variable including all inherited dimensions.

Offset
> The offset into the outermost parent structure.

Elemental size
> Elemental size is in bytes unless the variable is bit aligned, in which case it is in bits. In either case, this does not factor any in dimensions.

Size
> Size in bytes with its dimensions factored in.

Alignment
> Identified by the following:
> - 0 for bit-aligned
> - 7 for byte-aligned
> - 15 for halfword-aligned
> - 31 for fullword-aligned
> - 63 for quadword-aligned

A union within the record is dedicated to describing information that is dependent on the variable's storage class:

Static variables
> If the variable was declared as external with a separate external name (dcl x ext('y')), the literal index of that name is specified.

Based variables
> If the variable was declared as based on another mapped variable that is not an element of an array, the symbol index of that variable is specified.

Defined variables

If the variable was declared as defined on another mapped variable that is not an element of an array, the symbol index of that variable is specified here. If its position attribute is constant, it is also specified.

The variable's data type is specified by the ordinal shown in Figure 109.

```
define
  ordinal
    xin_Data_Kind
      (  xin_Data_Kind_Unset
        ,xin_Data_Kind_Character
        ,xin_Data_Kind_Bit
        ,xin_Data_Kind_Graphic
        ,xin_Data_Kind_Fixed
        ,xin_Data_Kind_Float
        ,xin_Data_Kind_Picture
        ,xin_Data_Kind_Pointer
        ,xin_Data_Kind_Offset
        ,xin_Data_Kind_Entry
        ,xin_Data_Kind_File
        ,xin_Data_Kind_Label
        ,xin_Data_Kind_Format
        ,xin_Data_Kind_Area
        ,xin_Data_Kind_Task
        ,xin_Data_Kind_Event
        ,xin_Data_Kind_Condition
        ,xin_Data_Kind_Structure
        ,xin_Data_Kind_Union
        ,xin_Data_Kind_Descriptor
        ,xin_Data_Kind_Ordinal
        ,xin_Data_Kind_Handle
        ,xin_Data_Kind_Type
                              )  prec(8) unsigned;
```

Figure 109. Data type of a variable

A union within the record is dedicated to describing information that is dependent on the variable's data type. Most of this information is self-explanatory (for example, the precision for an arithmetic type) except perhaps for the following:

Picture variables

The literal index of the picture specification is specified.

Entry variables

If the variable has the returns attribute, the symbol index of the returns description is specified.

Ordinal variables

The ordinal type index is specified.

Typed variables and handles

The symbol index of the underlying type is specified.

String and area variables

The type and value of the extent is specified in addition to the symbol index of the returns description. The type of the extent is encoded by the values:

```
declare
  (  xin_Extent_Constant     value(01)
    ,xin_Extent_Star         value(02)
    ,xin_Extent_Nonconstant  value(04)
```

```
      ,xin_Extent_Refer        value(08)
      ,xin_Extent_In_Error     value(16)
   )
   fixed bin;
```

If the element has any dimensions, the type and values for its lower and upper
bounds are specified at the very end of the record. These fields are not present if
the element has no dimensions. Figure 110 shows the declare for a symbol record.

```
declare                            /*                           */
  1 xin_Sym        based( null() ), /*                          */
                                   /*                           */
     2 Xin_Sym_Hdr                 /* standard header           */
          like Xin_Hdr,            /*                           */
                                   /*                           */
     2 Xin_Sym_File_Id             /* file id                   */
          fixed bin(32) unsigned,  /*                           */
                                   /*                           */
     2 Xin_Sym_Line_No             /* line no within file       */
          fixed bin(32) unsigned,  /*                           */
                                   /*                           */
     2 xin_Id                      /* identifying number        */
                  fixed bin(31),   /*                           */
                                   /*                           */
     2 xin_Sibling                 /* xin_id of next sibling    */
                  fixed bin(31),   /*                           */
                                   /*                           */
     2 xin_Parent                  /* xin_id of parent          */
                  fixed bin(31),   /*                           */
                                   /*                           */
     2 xin_Child                   /* xin_id of first child     */
                  fixed bin(31),   /*                           */
                                   /*                           */
     2 xin_Blk_Id                  /* blk_id of owning block    */
                  fixed bin(31),   /*                           */
                                   /*                           */
     2 xin_Sym_Tok                 /* token id of declaring token */
                  fixed bin(31),   /*                           */
                                   /*                           */
     2 xin_Logical_Level           /* logical level in structure */
          unsigned fixed bin(08),  /*                           */
                                   /*                           */
     2 xin_Physical_Level          /* given level in structure  */
          unsigned fixed bin(08),  /*                           */
                                   /*                           */
     2 xin_Total_Dims              /* Total number of dims      */
          unsigned fixed bin(08),  /*                           */
                                   /*                           */
     2 xin_Own_Dims                /* count of self-made dims   */
          unsigned fixed bin(08),  /*                           */
                                   /*                           */
```

Figure 110. Declare for a symbol record (Part 1 of 7)

```
     2 xin_Attr_Flags union,      /*                                    */
                                  /*                                    */
   3 *                bit(64),    /*                                    */
                                  /*                                    */
   3 *,                          /*                                    */
                                  /*                                    */
     4 xin_Attr_Automatic         /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Based             /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Controlled        /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Defined           /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Parameter         /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Position          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Reserved          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Static            /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Condition         /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Constant          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Variable          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Internal          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_External          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Abnormal          /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Normal            /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Assignable        /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Nonassignable     /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Aligned           /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Unaligned         /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Descriptor        /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Value             /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Byvalue           /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Byaddr            /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Connected         /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Nonconnected      /*                                    */
                     bit(1),      /*                                    */
     4 xin_Attr_Optional          /*                                    */
                     bit(1),      /*                                    */
```

Figure 110. Declare for a symbol record (Part 2 of 7)

```
               4 xin_Attr_Native         /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Nonnative      /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Initial        /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Typedef        /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Builtin        /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Generic        /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Date           /*                              */
                           bit(1),       /*                              */
               4 xin_Attr_Noinit         /*                              */
                           bit(1),       /*                              */
                                         /*                              */
         2 xin_Data_Is                   /*                              */
               ordinal xin_Data_Kind,    /*                              */
                                         /*                              */
         2 xin_Misc_Flags union,         /*                              */
                                         /*                              */
           3 *               bit(8),     /*                              */
                                         /*                              */
           3 *,                          /*                              */
                                         /*                              */
               4 xin_Implicit_Dcl        /* dcl is implicit              */
                           bit(1),       /*                              */
                                         /*                              */
               4 xin_Contextual_Dcl      /* dcl is contextual            */
                           bit(1),       /*                              */
                                         /*                              */
               4 xin_Has_Been_Mapped     /* aggregate has been mapped    */
                           bit(1),       /*                              */
                                         /*                              */
         2 xin_Align                     /* alignment                    */
               unsigned fixed bin(08),   /*                              */
                                         /*                              */
         2 xin_Begin_Offset              /* bitlocation(sym)             */
               unsigned fixed bin(08),   /*                              */
                                         /*                              */
         2 xin_Offset                    /* location(sym)                */
                       fixed bin(31),    /*                              */
                                         /*                              */
         2 xin_Size                      /* length in bytes, with all    */
                       fixed bin(31),    /*   children and array         */
                                         /*   elements factored in       */
                                         /*                              */
         2 xin_Base_Size                 /* element length - in bytes    */
                       fixed bin(31),    /*   unless bit aligned         */
                                         /*                              */
         2 xin_Name                      /* name - id of lit record      */
                       fixed bin(31),    /*                              */
                                         /*                              */
         2 * union,                      /*                              */
                                         /*                              */
           3 xin_Static_Data,            /*                              */
                                         /*                              */
               4 xin_Static_Ext          /* id of literal specifying its */
                       fixed bin(31),    /*   external name              */
```

Figure 110. Declare for a symbol record (Part 3 of 7)

```
                                   /*                          */
      3 xin_Based_Data,            /*                          */
                                   /*                          */
        4 xin_Based_On_Id          /* xin_Id of basing reference */
                fixed bin(31),     /*    0 if not simple       */
                                   /*                          */
      3 xin_Defined_Data,          /*                          */
                                   /*                          */
        4 xin_Defined_On_Id        /* xin_Id of basing reference */
                fixed bin(31),     /*    0 if not simple       */
                                   /*                          */
        4 xin_Defined_Pos          /* -1 if not constant       */
                fixed bin(31),     /*                          */
                                   /*                          */
      3 xin_Parm_Data,             /*                          */
                                   /*                          */
        4 xin_Parm_Index           /* index of parm            */
                fixed bin(31),     /*    1 for first, etc      */
                                   /*                          */
  2 * union,                       /*                          */
                                   /*                          */
    3 xin_Str_Data,                /* used for char, bit, graphic */
                                   /* and area, but not used for  */
                                   /* picture character or numeric*/
                                   /*                          */
      4 xin_Str_Len_Node           /* length as parse tree     */
                fixed bin(31),     /*                          */
                                   /*                          */
      4 xin_Str_Len                /* length: if type is constant */
                fixed bin(31),     /*                          */
                                   /*                          */
      4 xin_Str_Len_Type           /* type                     */
         unsigned fixed bin(08),   /*                          */
                                   /*                          */
      4 *,                         /*                          */
        5 xin_Str_Varying          /*                          */
                      bit(1),      /*                          */
        5 xin_Str_Nonvarying       /*                          */
                      bit(1),      /*                          */
        5 xin_Str_Varyingz         /*                          */
                      bit(1),      /*                          */
                                   /*                          */
      4 xin_Str_Date               /* index of date literal    */
                fixed bin(31),     /*                          */
                                   /*                          */
    3 xin_Arith_Data,              /* used for fixed and float */
                                   /*                          */
      4 xin_Arith_Precision        /* precision                */
         unsigned fixed bin(08),   /*                          */
                                   /*                          */
      4 xin_Arith_Scale_Factor     /* scale factor             */
          signed fixed bin(07),    /*                          */
                                   /*                          */
```

Figure 110. Declare for a symbol record (Part 4 of 7)

```
           4 *,                     /*                                      */
             5 xin_Arith_Binary     /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Decimal     /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Fixed       /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Float       /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Real        /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Complex     /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Signed      /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Unsigned    /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Ieee        /*                                      */
                          bit(1),    /*                                      */
             5 xin_Arith_Hexadec     /*                                      */
                          bit(1),    /*                                      */
                                     /*                                      */
           4 *                       /* unused                               */
                   fixed bin(31),    /*                                      */
                                     /*                                      */
           4 *                       /* unused                               */
                   fixed bin(31),    /*                                      */
                                     /*                                      */
           4 xin_Arith_Date          /* index of date literal                */
                   fixed bin(31),    /*                                      */
                                     /*                                      */
         3 xin_Ordinal_Data,         /* used for ordinal                     */
                                     /*                                      */
           4 xin_Ordinal_Type_Id     /* type id                              */
                   fixed bin(31),    /*                                      */
                                     /*                                      */
         3 xin_Type_Data,            /* used for typed                       */
                                     /*                                      */
           4 xin_Type_Is             /* type id                              */
                   fixed bin(31),    /*                                      */
                                     /*                                      */
         3 xin_Pic_Data,             /* used for all pictures                */
                                     /*                                      */
           4 xin_Pic_Ext             /* external specification               */
                   fixed bin(31),    /*                                      */
                                     /*                                      */
           4 *,                      /*                                      */
             5 xin_Pic_Fixed         /*                                      */
                          bit(1),    /*                                      */
             5 xin_Pic_Float         /*                                      */
                          bit(1),    /*                                      */
             5 xin_Pic_Character     /*                                      */
                          bit(1),    /*                                      */
             5 xin_Pic_Real          /*                                      */
                          bit(1),    /*                                      */
             5 xin_Pic_Complex       /*                                      */
                          bit(1),    /*                                      */
```

Figure 110. Declare for a symbol record (Part 5 of 7)

```
     5 *                    /*                          */
                bit(3),     /*                          */
     5 *                    /*                          */
                bit(8),     /*                          */
     5 xin_Pic_Prec         /*                          */
           unsigned         /*                          */
           fixed bin(08),   /*                          */
     5 xin_Pic_Scale        /*                          */
           signed           /*                          */
           fixed bin(07),   /*                          */
                            /*                          */
   4 *                      /* unused                   */
           fixed bin(31),   /*                          */
                            /*                          */
   4 xin_Pic_Date           /* index of date literal    */
           fixed bin(31),   /*                          */
                            /*                          */
 3 xin_Entry_Data,          /*                          */
                            /*                          */
   4 xin_Entry_Min          /* min number of args       */
           fixed bin(15),   /*   allowed when invoked   */
                            /*                          */
   4 xin_Entry_Max          /* max number of args       */
           fixed bin(15),   /*   allowed when invoked   */
                            /*                          */
   4 xin_Entry_Returns_Id   /* xin_Id of returns descriptor*/
           fixed bin(31),   /*                          */
                            /*                          */
   4 xin_Entry_Parms_Id     /* xin_Id of first parms    */
           fixed bin(31),   /*                          */
                            /*                          */
   4 *,                     /*                          */
     5 xin_Entry_Returns    /*                          */
                bit(1),     /*                          */
     5 xin_Entry_Limited    /*                          */
                bit(1),     /*                          */
     5 xin_Entry_Fetchable  /*                          */
                bit(1),     /*                          */
     5 xin_Entry_Is_Proc    /*                          */
                bit(1),     /*                          */
     5 xin_Entry_Is_Secondary /*                        */
                bit(1),     /*                          */
                            /*                          */
 3 xin_Ptr_Data,            /*                          */
                            /*                          */
   4 *,                     /*                          */
     5 xin_Ptr_Segmented    /*                          */
                bit(1),     /*                          */
                            /*                          */
 3 xin_Offset_Data,         /*                          */
                            /*                          */
   4 xin_Offset_Area        /*                          */
           fixed bin(31),   /*                          */
                            /*                          */
 3 xin_Sym_Bif_Id           /*                          */
       ordinal xin_Bif_Kind, /*                         */
                            /*                          */
```

Figure 110. Declare for a symbol record (Part 6 of 7)

```
                                     /*                            */
          3 xin_File_Data,           /*                            */
                                     /*                            */
            4 *,                     /*                            */
              5 xin_File_Buffered    /*                            */
                         bit(1),     /*                            */
              5 xin_File_Direct      /*                            */
                         bit(1),     /*                            */
              5 xin_File_Exclusive   /*                            */
                         bit(1),     /*                            */
              5 xin_File_Input       /*                            */
                         bit(1),     /*                            */
              5 xin_File_Keyed       /*                            */
                         bit(1),     /*                            */
              5 xin_File_Output      /*                            */
                         bit(1),     /*                            */
              5 xin_File_Print       /*                            */
                         bit(1),     /*                            */
              5 xin_File_Record      /*                            */
                         bit(1),     /*                            */
              5 xin_File_Stream      /*                            */
                         bit(1),     /*                            */
              5 xin_File_Transient   /*                            */
                         bit(1),     /*                            */
              5 xin_File_Unbuffered  /*                            */
                         bit(1),     /*                            */
              5 xin_File_Update      /*                            */
                         bit(1),     /*                            */
                                     /*                            */
          2 * union,                 /*                            */
                                     /*                            */
            3 xin_Value_Id           /* id of value lit - if the   */
                     fixed bin(31),  /*  xin_Attr_Value flag is set */
                                     /*                            */
            3 xin_First_Stmt_Id      /* id of first stmt record -  */
                     fixed bin(31),  /*  if xin_Attr_Entry and     */
                                     /*  xin_Entry_Is_Proc flags   */
                                     /*  are both set              */
                                     /*                            */
          2 xin_Bounds dim(15),      /*                            */
                                     /*                            */
            3 xin_Lbound_Type        /* lbound type                */
                unsigned fixed bin(08), /*                         */
                                     /*                            */
            3 xin_Hbound_Type        /* hbound type                */
                unsigned fixed bin(08), /*                         */
                                     /*                            */
            3 *                      /*                            */
                         char(2),    /*                            */
                                     /*                            */
            3 xin_Lbound_Node        /* expression parse tree      */
                     fixed bin(31),  /*                            */
                                     /*                            */
            3 xin_Hbound_Node        /* expression parse tree      */
                     fixed bin(31),  /*                            */
                                     /*                            */
            3 xin_Lbound             /* value:  if type is constant */
                     fixed bin(31),  /* xin_Id: if type is refer   */
                                     /*                            */
            3 xin_Hbound             /* value:  if type is constant */
                     fixed bin(31),  /* xin_Id: if type is refer   */
                                     /*                            */
          2 *            char(0);
```

Figure 110. Declare for a symbol record (Part 7 of 7)

The definition of the ordinal xin_Bif_Kind can be found in Figure 111

```
define
  ordinal
    xin_Bif_Kind
      (  xin_Bif_Unknown
        ,xin_bif_abs
        ,xin_bif_acos
        ,xin_bif_add
        ,xin_bif_addr
        ,xin_bif_all
        ,xin_bif_allocation
        ,xin_bif_allocn
        ,xin_bif_any
        ,xin_bif_asin
        ,xin_bif_atan
        ,xin_bif_atand
        ,xin_bif_atanh
        ,xin_bif_bin
        ,xin_bif_binvalue
        ,xin_bif_binary
        ,xin_bif_binaryvalue
        ,xin_bif_bit
        ,xin_bif_bool
        ,xin_bif_ceil
        ,xin_bif_char
        ,xin_bif_completion
        ,xin_bif_complex
        ,xin_bif_conjg
        ,xin_bif_copy
        ,xin_bif_cos
        ,xin_bif_cosd
        ,xin_bif_cosh
        ,xin_bif_count
        ,xin_bif_cpln
        ,xin_bif_cplx
        ,xin_bif_cstg
        ,xin_bif_currentstorage
        ,xin_bif_datafield
        ,xin_bif_date
        ,xin_bif_datetime
        ,xin_bif_dec
        ,xin_bif_decimal
        ,xin_bif_dim
        ,xin_bif_divide
        ,xin_bif_empty
        ,xin_bif_entryaddr
        ,xin_bif_erf
        ,xin_bif_erfc
        ,xin_bif_exp
        ,xin_bif_fixed
        ,xin_bif_float
        ,xin_bif_floor
        ,xin_bif_graphic
        ,xin_bif_hbound
```

Figure 111. Declare for xin_Bif_Kind (Part 1 of 5)

```
                        ,xin_bif_high
                        ,xin_bif_imag
                        ,xin_bif_index
                        ,xin_bif_lbound
                        ,xin_bif_length
                        ,xin_bif_lineno
                        ,xin_bif_log
                        ,xin_bif_log10
                        ,xin_bif_log2
                        ,xin_bif_low
                        ,xin_bif_max
                        ,xin_bif_min
                        ,xin_bif_mod
                        ,xin_bif_mpstr
                        ,xin_bif_multiply
                        ,xin_bif_null
                        ,xin_bif_offset
                        ,xin_bif_onchar
                        ,xin_bif_oncode
                        ,xin_bif_oncount
                        ,xin_bif_onfile
                        ,xin_bif_onkey
                        ,xin_bif_onloc
                        ,xin_bif_onsource
                        ,xin_bif_pageno
                        ,xin_bif_plicanc
                        ,xin_bif_plickpt
                        ,xin_bif_plidump
                        ,xin_bif_plirest
                        ,xin_bif_pliretc
                        ,xin_bif_pliretv
                        ,xin_bif_plisrta
                        ,xin_bif_plisrtb
                        ,xin_bif_plisrtc
                        ,xin_bif_plisrtd
                        ,xin_bif_plitest
                        ,xin_bif_pointer
                        ,xin_bif_pointeradd
                        ,xin_bif_pointervalue
                        ,xin_bif_poly
                        ,xin_bif_prec
                        ,xin_bif_precision
                        ,xin_bif_priority
                        ,xin_bif_prod
                        ,xin_bif_ptr
                        ,xin_bif_ptradd
                        ,xin_bif_ptrvalue
                        ,xin_bif_real
                        ,xin_bif_repeat
                        ,xin_bif_round
                        ,xin_bif_samekey
                        ,xin_bif_sign
                        ,xin_bif_sin
                        ,xin_bif_sind
                        ,xin_bif_sinh
                        ,xin_bif_sqrt
                        ,xin_bif_status
                        ,xin_bif_stg
```

Figure 111. Declare for xin_Bif_Kind (Part 2 of 5)

```
                    ,xin_bif_storage
                    ,xin_bif_string
                    ,xin_bif_substr
                    ,xin_bif_sum
                    ,xin_bif_sysnull
                    ,xin_bif_tan
                    ,xin_bif_tand
                    ,xin_bif_tanh
                    ,xin_bif_time
                    ,xin_bif_translate
                    ,xin_bif_trunc
                    ,xin_bif_unspec
                    ,xin_bif_verify
                    ,xin_bif_days
                    ,xin_bif_daystodate

                    ,xin_bif_acosf
                    ,xin_bif_addrdata
                    ,xin_bif_alloc
                    ,xin_bif_allocate
                    ,xin_bif_allocsize
                    ,xin_bif_asinf
                    ,xin_bif_atanf
                    ,xin_bif_auto
                    ,xin_bif_automatic
                    ,xin_bif_availablearea
                    ,xin_bif_bitloc
                    ,xin_bif_bitlocation
                    ,xin_bif_byte
                    ,xin_bif_cds
                    ,xin_bif_center
                    ,xin_bif_centerleft
                    ,xin_bif_centerright
                    ,xin_bif_centre
                    ,xin_bif_centreleft
                    ,xin_bif_centreright
                    ,xin_bif_character
                    ,xin_bif_charg
                    ,xin_bif_chargraphic
                    ,xin_bif_charval
                    ,xin_bif_checkstg
                    ,xin_bif_collate
                    ,xin_bif_compare
                    ,xin_bif_cosf
                    ,xin_bif_cs
                    ,xin_bif_currentsize
                    ,xin_bif_daystosecs
                    ,xin_bif_dimension
                    ,xin_bif_edit
                    ,xin_bif_endfile
                    ,xin_bif_epsilon
                    ,xin_bif_expf
                    ,xin_bif_exponent
                    ,xin_bif_fileddint
                    ,xin_bif_fileddtest
                    ,xin_bif_fileddword
                    ,xin_bif_fileid
                    ,xin_bif_fileread
                    ,xin_bif_fileseek
                    ,xin_bif_filetell
                    ,xin_bif_filewrite
                    ,xin_bif_gamma
                    ,xin_bif_getenv
```

Figure 111. Declare for xin_Bif_Kind (Part 3 of 5)

```
                    ,xin_bif_handle
                    ,xin_bif_hex
                    ,xin_bif_heximage
                    ,xin_bif_huge
                    ,xin_bif_iand
                    ,xin_bif_ieor
                    ,xin_bif_inot
                    ,xin_bif_ior
                    ,xin_bif_isigned
                    ,xin_bif_isll
                    ,xin_bif_ismain
                    ,xin_bif_isrl
                    ,xin_bif_iunsigned
                    ,xin_bif_left
                    ,xin_bif_loc
                    ,xin_bif_location
                    ,xin_bif_log10f
                    ,xin_bif_logf
                    ,xin_bif_loggamma
                    ,xin_bif_lower2
                    ,xin_bif_lowercase
                    ,xin_bif_maxexp
                    ,xin_bif_maxlength
                    ,xin_bif_memindex
                    ,xin_bif_memsearch
                    ,xin_bif_memsearchr
                    ,xin_bif_memverify
                    ,xin_bif_memverifyr
                    ,xin_bif_minexp
                    ,xin_bif_offsetadd
                    ,xin_bif_offsetdiff
                    ,xin_bif_offsetsubtract
                    ,xin_bif_offsetvalue
                    ,xin_bif_omitted
                    ,xin_bif_oncondcond
                    ,xin_bif_oncondid
                    ,xin_bif_ongsource
                    ,xin_bif_onsubcode
                    ,xin_bif_onwchar
                    ,xin_bif_onwsource
                    ,xin_bif_ordinalname
                    ,xin_bif_ordinalpred
                    ,xin_bif_ordinalsucc
                    ,xin_bif_packagename
                    ,xin_bif_picspec
                    ,xin_bif_places
                    ,xin_bif_pliascii
                    ,xin_bif_pliebcdic
                    ,xin_bif_plifill
                    ,xin_bif_plifree
                    ,xin_bif_plimove
                    ,xin_bif_pliover
                    ,xin_bif_plisaxa
                    ,xin_bif_plisaxb
```

Figure 111. Declare for xin_Bif_Kind (Part 4 of 5)

```
                ,xin_bif_pointerdiff
                ,xin_bif_pointersubtract
                ,xin_bif_pred
                ,xin_bif_present
                ,xin_bif_procedurename
                ,xin_bif_procname
                ,xin_bif_ptrdiff
                ,xin_bif_ptrsubtract
                ,xin_bif_putenv
                ,xin_bif_radix
                ,xin_bif_raise2
                ,xin_bif_random
                ,xin_bif_rank
                ,xin_bif_rem
                ,xin_bif_repattern
                ,xin_bif_replaceby2
                ,xin_bif_reverse
                ,xin_bif_right
                ,xin_bif_scale
                ,xin_bif_search
                ,xin_bif_searchr
                ,xin_bif_secs
                ,xin_bif_secstodate
                ,xin_bif_secstodays
                ,xin_bif_signed
                ,xin_bif_sinf
                ,xin_bif_size
                ,xin_bif_sourcefile
                ,xin_bif_sourceline
                ,xin_bif_sqrtf
                ,xin_bif_subtract
                ,xin_bif_succ
                ,xin_bif_system
                ,xin_bif_tally
                ,xin_bif_tanf
                ,xin_bif_threadid
                ,xin_bif_tiny
                ,xin_bif_trim
                ,xin_bif_type
                ,xin_bif_unallocated
                ,xin_bif_unsigned
                ,xin_bif_uppercase
                ,xin_bif_valid
                ,xin_bif_validdate
                ,xin_bif_varglist
                ,xin_bif_vargsize
                ,xin_bif_verifyr
                ,xin_bif_wchar
                ,xin_bif_wcharval
                ,xin_bif_weekday
                ,xin_bif_whigh
                ,xin_bif_widechar
                ,xin_bif_wlow
                ,xin_bif_xmlchar
                ,xin_bif_y4date
                ,xin_bif_y4julian
                ,xin_bif_y4year
        ) prec(16) unsigned;
```

Figure 111. Declare for xin_Bif_Kind (Part 5 of 5)

It should also be noted that the attributes flags reflect the attributes after the compiler has applied all defaults. So, for example, every numeric variable (including numeric PICTURE variables) will have either the REAL or COMPLEX attribute flag set.

Understanding SYSADATA syntax information

When you specify the SYN suboption of the XINFO compile-time option, the compiler generates a SYSADATA file that contains the following information in addition to the records generated for the MSG and SYM suboptions:

- Source records
- Token records
- Syntax records

Source records

Each source record assigns a number, called a source id, that is used by later records to refer to the source line described by this record. The line may be from a the primary PL/I source file or an INCLUDEd file, as indicated by the source file id and linenumber fields in the record. The rest of the record holds the actual data in the source line.

```
Dcl
  1 Xin_Src    Based( null() ),   /* source record                */
                                  /*                              */
    2 Xin_Src_Hdr                 /* standard header              */
         like Xin_Hdr,            /*                              */
                                  /*                              */
    2 Xin_Src_File_Id             /* file id                      */
         fixed bin(32) unsigned,  /*                              */
                                  /*                              */
    2 Xin_Src_Line_No             /* line no within file          */
         fixed bin(32) unsigned,  /*                              */
                                  /*                              */
    2 Xin_Src_Id                  /* id for this source record    */
         fixed bin(32) unsigned,  /*                              */
                                  /*                              */
    2 Xin_Src_Length              /* length of text               */
         fixed bin(16) unsigned,  /*                              */
                                  /*                              */
    2 Xin_Src_Text                /* actual text                  */
       char( 137 refer(xin_Src_Length) );
```

Figure 112. Declare for a source record

Token records

Each token record assigns a number, called a token index, that is used by later records to refer to a token recognized by the PL/I compiler. The record also identifies the type of the token plus the column and line on which it started and ended.

```
Dcl
  1 Xin_Tok     Based( null() ),  /* token record                */
                                  /*                             */
    2 Xin_Tok_Hdr                 /* standard header             */
          like Xin_Hdr,           /*                             */
                                  /*                             */
    2 Xin_Tok_Inx                 /* adata index for token       */
          fixed bin(32) unsigned, /*                             */
                                  /*                             */
    2 Xin_Tok_Begin_Line          /* starting line no within file */
          fixed bin(32) unsigned, /*                             */
                                  /*                             */
    2 Xin_Tok_End_Line_Offset     /* offset of end line from first */
          fixed bin(16) unsigned, /*                             */
                                  /*                             */
    2 Xin_Tok_Kind_Value          /* token kind                  */
          ordinal xin_Tok_Kind,   /*                             */
                                  /*                             */
    2 Xin_Tok_Rsrvd               /* reserved                    */
          fixed bin(8) unsigned,  /*                             */
                                  /*                             */
    2 Xin_Tok_Begin_Col           /* starting column             */
          fixed bin(16) unsigned, /*                             */
                                  /*                             */
    2 Xin_Tok_End_Col             /* ending column               */
          fixed bin(16) unsigned; /*                             */
```

Figure 113. Declare for a token record

The ordinal xin_Tok_Kind identifies the type of the token record.

```
Define
 ordinal
   xin_Tok_Kind
     (  xin_Tok_Kind_Unset
       ,xin_Tok_Kind_Lexeme
       ,xin_Tok_Kind_Comment
       ,xin_Tok_Kind_Literal
       ,xin_Tok_Kind_Identifier
       ,xin_Tok_Kind_Keyword
                             )  prec(8) unsigned;
```

Figure 114. Declare for the token record kind

Syntax records

Each syntax record assigns a number, called a node id, that is used by later records to refer to other syntax records.

The first syntax record will have kind xin_Syn_Kind_Package, and if the compilation unit has any procedures, the child node of this record will point to the first of these procedures. The parent, sibling and child nodes will then provide a map with the appropriate relationships of all the procedures and begin bocks in the compilation unit.

Consider the following simple program:

```
a: proc;
  call b;
  call c;
  b: proc;
```

```
      end b;
    c: proc;
      call d;
      d: proc;
      end d;
    end c;
  end a;
```

The node indices assigned to the blocks of the preceding program would be as follows:

```
symbol   index    sibling     parent    child
-----    -----    -------     ------    -----
  -        1         0           0        2
  a        2         0           1        3
  b        3         4           2        0
  c        4         0           2        5
  d        5         0           4        0
```

Figure 115. Node indices assigned to the blocks in a program

```
Dcl
  1 Xin_Syn      Based( null() ),  /* syntax record              */
                                   /*                            */
    2 Xin_Syn_Hdr                  /* standard header            */
         like Xin_Hdr,             /*                            */
                                   /*                            */
    2 Xin_Syn_Node_Id             /* node id                    */
         fixed bin(32) unsigned,   /*                            */
                                   /*                            */
    2 Xin_Syn_Node_Kind           /* node type                  */
         ordinal xin_syn_kind,     /*                            */
                                   /*                            */
    2 Xin_Syn_Node_Exp_Kind       /* node sub type              */
         ordinal xin_exp_kind,     /*                            */
                                   /*                            */
    2 *                            /* reserved                   */
         fixed bin(16) unsigned,   /*                            */
                                   /*                            */
    2 Xin_Syn_Parent_Node_Id      /* node id of parent          */
         fixed bin(32) unsigned,   /*                            */
                                   /*                            */
    2 Xin_Syn_Sibling_Node_Id     /* node id of sibling         */
         fixed bin(32) unsigned,   /*                            */
                                   /*                            */
    2 Xin_Syn_Child_Node_Id       /* node id of child           */
         fixed bin(32) unsigned,   /*                            */
                                   /*                            */
    2 xin_Syn_First_Tok           /* id of first spanned token  */
         fixed bin(32) unsigned,   /*                            */
                                   /*                            */
    2 xin_Syn_Last_Tok            /* id of last spanned token   */
         fixed bin(32) unsigned,   /*                            */
```

Figure 116. Declare for a syntax record (Part 1 of 3)

```
                                  /*                              */
    2 * union,                    /* qualifier for node          */
                                  /*                              */
      3 Xin_Syn_Int_Value         /* used if int                 */
          fixed bin(31),          /*                              */
                                  /*                              */
      3 Xin_Syn_Literal_Id        /* used if name, number, picture */
          fixed bin(31),          /*                              */
                                  /*                              */
      3 Xin_Syn_Node_Lex          /* used if lexeme, assignment,  */
          ordinal xin_Lex_kind,   /* infix_op, prefix_op          */
                                  /*                              */
      3 Xin_Syn_Node_Voc          /* used if keyword, end_for_do  */
          ordinal xin_Voc_kind,   /*                              */
                                  /*                              */
      3 Xin_Syn_Block_Node        /* used if call_begin           */
          fixed bin(31),          /*   to hold node of begin block */
                                  /*                              */
      3 Xin_Syn_Bif_Id            /* used if bif_rfrnc            */
          fixed bin(32) unsigned, /*                              */
                                  /*                              */
      3 Xin_Syn_Sym_Id            /* used if label, unsub_rfrnc,  */
          fixed bin(32) unsigned, /* subscripted_rfrnc            */
                                  /*                              */
      3 Xin_Syn_Proc_Data,        /* used if package, proc or begin */
                                  /*                              */
        4 Xin_Syn_First_Sym       /*   id of first contained sym  */
            fixed bin(32) unsigned, /*                            */
                                  /*                              */
        4 Xin_Syn_Block_Sym       /*   id of sym for this block   */
            fixed bin(32) unsigned, /*                            */
```

Figure 116. Declare for a syntax record (Part 2 of 3)

```
                                   /*                               */
        3 Xin_Syn_Number_Data,     /* used if number                */
                                   /*                               */
          4 Xin_Syn_Number_Id      /*   id of literal               */
             fixed bin(32) unsigned, /*                             */
                                   /*                               */
          4 Xin_Syn_Number_Type    /*   type                        */
             ordinal xin_Number_Kind,/*                             */
                                   /*                               */
          4 Xin_Syn_Number_Prec    /*   precision                   */
             fixed bin(8) unsigned, /*                              */
                                   /*                               */
          4 Xin_Syn_Number_Scale   /*   scale factor                */
             fixed bin(7) signed,   /*                              */
                                   /*                               */
          4 Xin_Syn_Number_Bytes   /*   bytes it would occupy       */
             fixed bin(8) unsigned, /*   in its internal form       */
                                   /*                               */
        3 Xin_Syn_String_Data,     /* used if char_string,          */
                                   /* bit_string, graphic_string    */
                                   /*                               */
          4 Xin_Syn_String_Id      /*   id of literal               */
             fixed bin(32) unsigned, /*                             */
                                   /*                               */
          4 Xin_Syn_String_Len     /*   string length in its units  */
             fixed bin(32) unsigned, /*                             */
                                   /*                               */
        3 Xin_Syn_Stmt_Data,       /* used if stmt                  */
                                   /*                               */
          4 Xin_Syn_File_Id        /*   file id                     */
             fixed bin(32) unsigned, /*                             */
                                   /*                               */
          4 Xin_Syn_Line_No        /*   line no within file         */
             fixed bin(32) unsigned, /*                             */
                                   /*                               */
      2 *              char(0);    /*                               */
```

Figure 116. Declare for a syntax record (Part 3 of 3)

The ordinal xin_Syn_Kind identifies the type of the syntax record.

```
Define
  ordinal
    xin_Syn_Kind
      (   xin_Syn_Kind_Unset
        ,xin_Syn_Kind_Lexeme
        ,xin_Syn_Kind_Asterisk
        ,xin_Syn_Kind_Int
        ,xin_Syn_Kind_Name
        ,xin_Syn_Kind_Expression
        ,xin_Syn_Kind_Parenthesized_Expr
        ,xin_Syn_Kind_Argument_List
        ,xin_Syn_Kind_Keyword
        ,xin_Syn_Kind_Proc_Stmt
        ,xin_Syn_Kind_Begin_Stmt
        ,xin_Syn_Kind_Stmt
        ,xin_Syn_Kind_Substmt
        ,xin_Syn_Kind_Label
        ,xin_Syn_Kind_Invoke_Begin
        ,xin_Syn_Kind_Assignment
        ,xin_Syn_Kind_Assignment_Byname
        ,xin_Syn_Kind_Do_Fragment
        ,xin_Syn_Kind_Keyed_List
        ,xin_Syn_Kind_Iteration_Factor
        ,xin_Syn_Kind_If_Clause
        ,xin_Syn_Kind_Else_Clause
        ,xin_Syn_Kind_Do_Stmt
        ,xin_Syn_Kind_Select_Stmt
        ,xin_Syn_Kind_When_Stmt
        ,xin_Syn_Kind_Otherwise_Stmt
        ,xin_Syn_Kind_Procedure
        ,xin_Syn_Kind_Package
        ,xin_Syn_Kind_Begin_Block
        ,xin_Syn_Kind_Picture
        ,xin_Syn_Kind_Raw_Rfrnc
        ,xin_Syn_Kind_Generic_Desc
                              )  prec(8) unsigned;
```

Figure 117. Declare for the syntax record kind

Consider the following simple program:

```
a: proc(x);
   dcl x char(8);
   x = substr(datetime(),1,8);
end;
```

The node indices assigned to the blocks of the preceding program would be as
follows:

```
node_kind      index    sibling    parent    child
-----------    -----    -------    ------    -----
 package         1         0          0        2
 procedure       2         0          1        0
 expression      3         0          0        0
 stmt            4         5          2        6
 stmt            5        10          2       11
 label           6         7          4        0
 keyword         7         8          4        0
 expression      8         9          4        0
 lexeme          9         0          4        0
 stmt           10         0          2       18
 assignment     11        12          5       13
 lexeme         12         0          5        0
 expression     13        14         11        0
 expression     14         0         11       15
 expression     15        16         14        0
 expression     16        17         14        0
 expression     17         0         14        0
 keyword        18        19         10        0
 lexeme         19         0         10        0
```

Figure 118. Node indices assigned to the syntax records in a program

The procedure record contains the identifier (in the block_sym field) for the symbol record for ENTRY A. This symbol record contains, in turn, the node identifier (in the first_stmt_id field) for the first statement in that procedure.

Note that for the statement records
- the sibling node identifier points to the next statement record, if any
- the child node identifier points to the first element of that statement record

The records for the PROCEDURE statement consists of 4 records:
- a label record
- a keyword record (for the PROCEDURE keyword)
- an expression record (for the parameter X) with expression kind of unsub_rfrnc and a sym_id for the symbol X
- a lexeme record (for the semicolon)

The records for the assignment statement consists of 2 records:
- an assignment record which has 2 children:
  - an expression record (for the target X) with expression kind of unsub_rfrnc and a sym_id for the symbol X
  - an expression record (for the source) with expression kind of builtin_rfrnc and a sym_id for the symbol SUBSTR, and this record has itself 3 children:
    - an expression record (for the first argument) with expression kind of builtin_rfrnc and a sym_id for the symbol DATETIME
    - an expression record (for the second argument) with expression kind of number and a literal_id for the value 1
    - an expression record (for the third argument) with expression kind of number and a literal_id for the value 8
- a lexeme record (for the semicolon)

The records for the END statement consists of 2 records:

- a keyword record (for the END keyword)
- a lexeme record (for the semicolon)

The ordinal xin_Exp_Kind identifies the type of an expression for a syntax record that describes an expression. Some of these records will have non-zero child nodes, for example:

- an infix op, such as a minus for a subtraction, will have a child node that describes its lefthand operand (and the sibling node of that operand will describe the righthand operator)
- a prefix op, such as a minus for a negation, will have a child node that describes its operand

```
Define
 ordinal
   xin_Exp_Kind
     (  xin_Exp_Kind_Unset
       ,xin_Exp_Kind_Bit_String
       ,xin_Exp_Kind_Char_String
       ,xin_Exp_Kind_Graphic_String
       ,xin_Exp_Kind_Number
       ,xin_Exp_Kind_Infix_Op
       ,xin_Exp_Kind_Prefix_Op
       ,xin_Exp_Kind_Builtin_Rfrnc
       ,xin_Exp_Kind_Entry_Rfrnc
       ,xin_Exp_Kind_Qualified_Rfrnc
       ,xin_Exp_Kind_Unsub_Rfrnc
       ,xin_Exp_Kind_Subscripted_Rfrnc
       ,xin_Exp_Kind_Type_Func
       ,xin_Exp_Kind_Widechar_String
                                   )  prec(8) unsigned;
```

Figure 119. Declare for the expression kind

The ordinal xin_Number_Kind identifies the type of a number for a syntax record that describes a number.

```
Define
 ordinal
   xin_Number_Kind
     (  xin_Number_Kind_Unset
       ,xin_Number_Kind_Real_Fixed_Bin
       ,xin_Number_Kind_Real_Fixed_Dec
       ,xin_Number_Kind_Real_Float_Bin
       ,xin_Number_Kind_Real_Float_Dec
       ,xin_Number_Kind_Cplx_Fixed_Bin
       ,xin_Number_Kind_Cplx_Fixed_Dec
       ,xin_Number_Kind_Cplx_Float_Bin
       ,xin_Number_Kind_Cplx_Float_Dec
                                   )  prec(8) unsigned;
```

Figure 120. Declare for the number kind

The ordinal xin_Lex_Kind identifies the type of a lexeme for a syntax record that describes a lexical unit. In these ordinal names,
- "vrule" means "vertical rule" which is used, for instance, as the "or" symbol

- "dbl" means "double", so that dbl_Vrule is a doubled vertical rule which is used, for instance, as the "concatenate" symbol

```
Define
 ordinal
    xin_Lex_Kind
       (  xin_Lex_Undefined
         ,xin_Lex_Period
         ,xin_Lex_Colon
         ,xin_Lex_Semicolon
         ,xin_Lex_Lparen
         ,xin_Lex_Rparen
         ,xin_Lex_Comma
         ,xin_Lex_Equals
         ,xin_Lex_Gt
         ,xin_Lex_Ge
         ,xin_Lex_Lt
         ,xin_Lex_Le
         ,xin_Lex_Ne
         ,xin_Lex_Lctr
         ,xin_Lex_Star
         ,xin_Lex_Dbl_Colon
         ,xin_Lex_Not
         ,xin_Lex_Vrule
         ,xin_Lex_Dbl_Vrule
         ,xin_Lex_And
         ,xin_Lex_Dbl_Star
         ,xin_Lex_Plus
         ,xin_Lex_Minus
         ,xin_Lex_Slash
         ,xin_Lex_Equals_Gt
         ,xin_Lex_Lparen_Colon
         ,xin_Lex_Colon_Rparen
         ,xin_Lex_Plus_Equals
         ,xin_Lex_Minus_Equals
         ,xin_Lex_Star_Equals
         ,xin_Lex_Slash_Equals
         ,xin_Lex_Vrule_Equals
         ,xin_Lex_And_Equals
         ,xin_Lex_Dbl_Star_Equals
         ,xin_Lex_Dbl_Vrule_Equals
         ,xin_Lex_Dbl_Slash
       ) unsigned prec(16);
```

Figure 121. Declare for the lexeme kind

The ordinal xin_Voc_Kind identifies the keyword for a syntax record that describes an item from the compiler's "vocabulary".

```
Define
  ordinal
    xin_Voc_Kind
      (  xin_Voc_Undefined
        ,xin_Voc_a
        ,xin_Voc_abnormal
        ,xin_Voc_act
        ,xin_Voc_activate
        ,xin_Voc_adata
        ,xin_Voc_addbuff
        ,xin_Voc_aggregate
        ,xin_Voc_aix
        ,xin_Voc_alias
        ,xin_Voc_alien
        ,xin_Voc_aligned
        ,xin_Voc_all
        ,xin_Voc_alloc
        ,xin_Voc_allocate
        ,xin_Voc_anno
        ,xin_Voc_ans
        ,xin_Voc_any
        ,xin_Voc_anycond
        ,xin_Voc_anycondition
        ,xin_Voc_area
        ,xin_Voc_as
        ,xin_Voc_ascii
        ,xin_Voc_asgn
        ,xin_Voc_asm
        ,xin_Voc_asmtdli
        ,xin_Voc_assembler
        ,xin_Voc_assignable
        ,xin_Voc_attach
        ,xin_Voc_attention
        ,xin_Voc_attn
        ,xin_Voc_attribute
        ,xin_Voc_attributes
        ,xin_Voc_auto
        ,xin_Voc_automatic
        ,xin_Voc_b
        ,xin_Voc_backwards
        ,xin_Voc_based
        ,xin_Voc_begin
        ,xin_Voc_beta
        ,xin_Voc_bigendian
        ,xin_Voc_bin
        ,xin_Voc_binary
        ,xin_Voc_bind
        ,xin_Voc_bit
        ,xin_Voc_bkwd
        ,xin_Voc_blksize
        ,xin_Voc_block
        ,xin_Voc_buf
```

Figure 122. Declare for the voc kind (Part 1 of 15)

```
                  ,xin_Voc_buffered
                  ,xin_Voc_buffers
                  ,xin_Voc_bufnd
                  ,xin_Voc_bufni
                  ,xin_Voc_bufoff
                  ,xin_Voc_bufsp
                  ,xin_Voc_build
                  ,xin_Voc_builtin
                  ,xin_Voc_by
                  ,xin_Voc_byaddr
                  ,xin_Voc_byname
                  ,xin_Voc_byvalue
                  ,xin_Voc_c
                  ,xin_Voc_call
                  ,xin_Voc_cdecl
                  ,xin_Voc_cdecl16
                  ,xin_Voc_cee
                  ,xin_Voc_ceetdli
                  ,xin_Voc_cell
                  ,xin_Voc_char
                  ,xin_Voc_character
                  ,xin_Voc_charg
                  ,xin_Voc_chargraphic
                  ,xin_Voc_charset
                  ,xin_Voc_check
                  ,xin_Voc_cics
                  ,xin_Voc_class
                  ,xin_Voc_close
                  ,xin_Voc_cmp
                  ,xin_Voc_cmpat
                  ,xin_Voc_cms
                  ,xin_Voc_cmstpl
                  ,xin_Voc_cobol
                  ,xin_Voc_col
                  ,xin_Voc_column
                  ,xin_Voc_compile
                  ,xin_Voc_complex
                  ,xin_Voc_cond
                  ,xin_Voc_condition
                  ,xin_Voc_conn
                  ,xin_Voc_connected
                  ,xin_Voc_consecutive
                  ,xin_Voc_constant
                  ,xin_Voc_control
                  ,xin_Voc_controlled
                  ,xin_Voc_conv
                  ,xin_Voc_conversion
                  ,xin_Voc_copy
                  ,xin_Voc_count
                  ,xin_Voc_cplx
```

Figure 122. Declare for the voc kind (Part 2 of 15)

```
                ,xin_Voc_create
                ,xin_Voc_cs
                ,xin_Voc_ct
                ,xin_Voc_ctl
                ,xin_Voc_ctl360
                ,xin_Voc_ctlasa
                ,xin_Voc_currency
                ,xin_Voc_d
                ,xin_Voc_data
                ,xin_Voc_dataonly
                ,xin_Voc_db
                ,xin_Voc_dcl
                ,xin_Voc_deact
                ,xin_Voc_deactivate
                ,xin_Voc_debug
                ,xin_Voc_dec
                ,xin_Voc_decimal
                ,xin_Voc_deck
                ,xin_Voc_declare
                ,xin_Voc_def
                ,xin_Voc_default
                ,xin_Voc_define
                ,xin_Voc_defined
                ,xin_Voc_defines
                ,xin_Voc_delay
                ,xin_Voc_delete
                ,xin_Voc_desclist
                ,xin_Voc_desclocator
                ,xin_Voc_descriptor
                ,xin_Voc_descriptors
                ,xin_Voc_detach
                ,xin_Voc_dft
                ,xin_Voc_dim
                ,xin_Voc_dimension
                ,xin_Voc_direct
                ,xin_Voc_directed
                ,xin_Voc_display
                ,xin_Voc_dli
                ,xin_Voc_dllinit
                ,xin_Voc_do
                ,xin_Voc_downthru
                ,xin_Voc_dummydesc
                ,xin_Voc_duplicate
                ,xin_Voc_e
                ,xin_Voc_ebcdic
                ,xin_Voc_edit
                ,xin_Voc_alpha
                ,xin_Voc_else
                ,xin_Voc_emulate
                ,xin_Voc_enclave
```

Figure 122. Declare for the voc kind (Part 3 of 15)

```
               ,xin_Voc_end
               ,xin_Voc_endf
               ,xin_Voc_endfile
               ,xin_Voc_endif
               ,xin_Voc_endp
               ,xin_Voc_endpage
               ,xin_Voc_entry
               ,xin_Voc_enu
               ,xin_Voc_env
               ,xin_Voc_environment
               ,xin_Voc_error
               ,xin_Voc_esd
               ,xin_Voc_evendec
               ,xin_Voc_event
               ,xin_Voc_exclusive
               ,xin_Voc_exec
               ,xin_Voc_execops
               ,xin_Voc_execute
               ,xin_Voc_exit
               ,xin_Voc_exports
               ,xin_Voc_ext
               ,xin_Voc_extchk
               ,xin_Voc_external
               ,xin_Voc_externalonly
               ,xin_Voc_extname
               ,xin_Voc_extonly
               ,xin_Voc_f
               ,xin_Voc_fastcall
               ,xin_Voc_fastcall16
               ,xin_Voc_fb
               ,xin_Voc_fbs
               ,xin_Voc_fetch
               ,xin_Voc_fetchable
               ,xin_Voc_file
               ,xin_Voc_finish
               ,xin_Voc_first
               ,xin_Voc_fixed
               ,xin_Voc_fixeddec
               ,xin_Voc_fixedoverflow
               ,xin_Voc_flag
               ,xin_Voc_float
               ,xin_Voc_flow
               ,xin_Voc_flush
               ,xin_Voc_fofl
               ,xin_Voc_forever
               ,xin_Voc_format
               ,xin_Voc_fortran
```

Figure 122. Declare for the voc kind (Part 4 of 15)

```
             ,xin_Voc_free
             ,xin_Voc_from
             ,xin_Voc_fromalien
             ,xin_Voc_fs
             ,xin_Voc_full
             ,xin_Voc_g
             ,xin_Voc_generic
             ,xin_Voc_genkey
             ,xin_Voc_get
             ,xin_Voc_gn
             ,xin_Voc_go
             ,xin_Voc_gonumber
             ,xin_Voc_gostmt
             ,xin_Voc_goto
             ,xin_Voc_gr
             ,xin_Voc_graphic
             ,xin_Voc_gs
             ,xin_Voc_halt
             ,xin_Voc_handle
             ,xin_Voc_hexadec
             ,xin_Voc_hexadecimal
             ,xin_Voc_i
             ,xin_Voc_ibm
             ,xin_Voc_ieee
             ,xin_Voc_if
             ,xin_Voc_ign
             ,xin_Voc_ignore
             ,xin_Voc_imp
             ,xin_Voc_impl
             ,xin_Voc_implicit
             ,xin_Voc_imported
             ,xin_Voc_imprecise
             ,xin_Voc_ims
             ,xin_Voc_in
             ,xin_Voc_inc
             ,xin_Voc_incafter
             ,xin_Voc_incdir
             ,xin_Voc_include
             ,xin_Voc_incpath
             ,xin_Voc_indexarea
             ,xin_Voc_indexed
             ,xin_Voc_inherits
             ,xin_Voc_init
             ,xin_Voc_initfill
             ,xin_Voc_initial
             ,xin_Voc_inline
             ,xin_Voc_inout
```

Figure 122. Declare for the voc kind (Part 5 of 15)

```
             ,xin_Voc_input
             ,xin_Voc_insource
             ,xin_Voc_instance
             ,xin_Voc_int
             ,xin_Voc_inter
             ,xin_Voc_internal
             ,xin_Voc_interrupt
             ,xin_Voc_into
             ,xin_Voc_invalidop
             ,xin_Voc_ipa
             ,xin_Voc_irred
             ,xin_Voc_irreducible
             ,xin_Voc_is
             ,xin_Voc_iterate
             ,xin_Voc_itrace
             ,xin_Voc_jpn
             ,xin_Voc_k
             ,xin_Voc_key
             ,xin_Voc_keyed
             ,xin_Voc_keyfrom
             ,xin_Voc_keylength
             ,xin_Voc_keyloc
             ,xin_Voc_keyto
             ,xin_Voc_l
             ,xin_Voc_label
             ,xin_Voc_langlvl
             ,xin_Voc_last
             ,xin_Voc_laxconv
             ,xin_Voc_laxdcl
             ,xin_Voc_laxif
             ,xin_Voc_laxint
             ,xin_Voc_laxqual
             ,xin_Voc_lc
             ,xin_Voc_leave
             ,xin_Voc_library
             ,xin_Voc_libs
             ,xin_Voc_like
             ,xin_Voc_limited
             ,xin_Voc_limits
             ,xin_Voc_line
             ,xin_Voc_linecount
             ,xin_Voc_lineno
             ,xin_Voc_linesize
             ,xin_Voc_linkage
             ,xin_Voc_list
             ,xin_Voc_littleendian
             ,xin_Voc_lmessage
             ,xin_Voc_lmsg
             ,xin_Voc_local
             ,xin_Voc_localonly
```

Figure 122. Declare for the voc kind (Part 6 of 15)

```
,xin_Voc_locate
,xin_Voc_log
,xin_Voc_loop
,xin_Voc_lowerinc
,xin_Voc_lsfirst
,xin_Voc_m
,xin_Voc_macro
,xin_Voc_main
,xin_Voc_map
,xin_Voc_mar
,xin_Voc_margini
,xin_Voc_margins
,xin_Voc_mask
,xin_Voc_max
,xin_Voc_maxgen
,xin_Voc_maxmem
,xin_Voc_md
,xin_Voc_mdeck
,xin_Voc_member
,xin_Voc_metaclass
,xin_Voc_method
,xin_Voc_methods
,xin_Voc_mi
,xin_Voc_min
,xin_Voc_msfirst
,xin_Voc_msg
,xin_Voc_multi
,xin_Voc_mvs
,xin_Voc_n
,xin_Voc_na
,xin_Voc_nag
,xin_Voc_name
,xin_Voc_names
,xin_Voc_nan
,xin_Voc_native
,xin_Voc_nativeaddr
,xin_Voc_natlang
,xin_Voc_nc
,xin_Voc_ncp
,xin_Voc_nct
,xin_Voc_nd
,xin_Voc_nest
,xin_Voc_new
,xin_Voc_ngn
,xin_Voc_ngr
,xin_Voc_ngs
,xin_Voc_nign
,xin_Voc_nimp
,xin_Voc_nimpl
,xin_Voc_ninc
```

Figure 122. Declare for the voc kind (Part 7 of 15)

```
                    ,xin_Voc_nint
                    ,xin_Voc_nis
                    ,xin_Voc_nm
                    ,xin_Voc_nmd
                    ,xin_Voc_nmi
                    ,xin_Voc_nnum
                    ,xin_Voc_noadata
                    ,xin_Voc_noaggregate
                    ,xin_Voc_noanno
                    ,xin_Voc_noattributes
                    ,xin_Voc_noauto
                    ,xin_Voc_noautomatic
                    ,xin_Voc_nobj
                    ,xin_Voc_nobuild
                    ,xin_Voc_nocee
                    ,xin_Voc_nocharg
                    ,xin_Voc_nochargraphic
                    ,xin_Voc_nocheck
                    ,xin_Voc_nocompile
                    ,xin_Voc_noconv
                    ,xin_Voc_noconversion
                    ,xin_Voc_nocount
                    ,xin_Voc_nodebug
                    ,xin_Voc_nodeck
                    ,xin_Voc_nodef
                    ,xin_Voc_nodescriptor
                    ,xin_Voc_nodescriptors
                    ,xin_Voc_nodirected
                    ,xin_Voc_nodli
                    ,xin_Voc_nodllinit
                    ,xin_Voc_nodummydesc
                    ,xin_Voc_noduplicate
                    ,xin_Voc_noemulate
                    ,xin_Voc_noesd
                    ,xin_Voc_noevendec
                    ,xin_Voc_noexecops
                    ,xin_Voc_noexit
                    ,xin_Voc_noext
                    ,xin_Voc_noextchk
                    ,xin_Voc_nof
                    ,xin_Voc_nofetchable
                    ,xin_Voc_nofixedoverflow
                    ,xin_Voc_noflow
                    ,xin_Voc_nofofl
                    ,xin_Voc_nofromalien
                    ,xin_Voc_nogonumber
                    ,xin_Voc_nogostmt
                    ,xin_Voc_nographic
                    ,xin_Voc_noignore
                    ,xin_Voc_noimplicit
```

Figure 122. Declare for the voc kind (Part 8 of 15)

```
,xin_Voc_noimprecise
,xin_Voc_noinclude
,xin_Voc_noinitfill
,xin_Voc_noinline
,xin_Voc_noinsource
,xin_Voc_nointerrupt
,xin_Voc_noinvalidop
,xin_Voc_noipa
,xin_Voc_nolaxasgn
,xin_Voc_nolaxconv
,xin_Voc_nolaxdcl
,xin_Voc_nolaxif
,xin_Voc_nolaxint
,xin_Voc_nolaxqual
,xin_Voc_nolibs
,xin_Voc_nolist
,xin_Voc_nolock
,xin_Voc_nolog
,xin_Voc_nomacro
,xin_Voc_nomap
,xin_Voc_nomapin
,xin_Voc_nomapout
,xin_Voc_nomargini
,xin_Voc_nomdeck
,xin_Voc_nomsg
,xin_Voc_nonasgn
,xin_Voc_nonassignable
,xin_Voc_nonconn
,xin_Voc_nonconnected
,xin_Voc_none
,xin_Voc_nonest
,xin_Voc_nonlocal
,xin_Voc_nonnative
,xin_Voc_nonnativeaddr
,xin_Voc_nonrecursive
,xin_Voc_nonumber
,xin_Voc_nonvar
,xin_Voc_nonvarying
,xin_Voc_noobject
,xin_Voc_nooffset
,xin_Voc_noofl
,xin_Voc_nooptimize
,xin_Voc_nooptions
,xin_Voc_nooverflow
,xin_Voc_nop
,xin_Voc_nopp
,xin_Voc_nopptrace
,xin_Voc_noprobe
,xin_Voc_noproceed
,xin_Voc_noprofile
```

Figure 122. Declare for the voc kind (Part 9 of 15)

```
,xin_Voc_nopt
,xin_Voc_norb
,xin_Voc_noreserve
,xin_Voc_noretcode
,xin_Voc_normal
,xin_Voc_norunops
,xin_Voc_noscheduler
,xin_Voc_nosemantic
,xin_Voc_nosequence
,xin_Voc_noshort
,xin_Voc_nosize
,xin_Voc_nosnap
,xin_Voc_nosource
,xin_Voc_nosprog
,xin_Voc_nostmt
,xin_Voc_nostorage
,xin_Voc_nostrg
,xin_Voc_nostringrange
,xin_Voc_nostringsize
,xin_Voc_nostrz
,xin_Voc_nosubrg
,xin_Voc_nosubscriptrang
,xin_Voc_nosym
,xin_Voc_nosyntax
,xin_Voc_not
,xin_Voc_noterminal
,xin_Voc_notest
,xin_Voc_notiled
,xin_Voc_notrace
,xin_Voc_noufl
,xin_Voc_nounderflow
,xin_Voc_nowcode
,xin_Voc_nowrite
,xin_Voc_noxref
,xin_Voc_nozdiv
,xin_Voc_nozerodivide
,xin_Voc_npro
,xin_Voc_ns
,xin_Voc_nsem
,xin_Voc_nseq
,xin_Voc_nstg
,xin_Voc_nsyn
,xin_Voc_nterm
,xin_Voc_null370
,xin_Voc_nullsys
,xin_Voc_num
,xin_Voc_number
,xin_Voc_nx
,xin_Voc_obj
,xin_Voc_object
,xin_Voc_of
,xin_Voc_offset
,xin_Voc_ofl
,xin_Voc_on
,xin_Voc_onproc
,xin_Voc_op
```

Figure 122. Declare for the voc kind (Part 10 of 15)

```
,xin_Voc_open
,xin_Voc_opt
,xin_Voc_optimize
,xin_Voc_optional
,xin_Voc_options
,xin_Voc_optlink
,xin_Voc_or
,xin_Voc_order
,xin_Voc_ordinal
,xin_Voc_organization
,xin_Voc_os
,xin_Voc_os2
,xin_Voc_other
,xin_Voc_otherwise
,xin_Voc_out
,xin_Voc_output
,xin_Voc_overflow
,xin_Voc_overrides
,xin_Voc_owns
,xin_Voc_p
,xin_Voc_package
,xin_Voc_page
,xin_Voc_pagesize
,xin_Voc_parameter
,xin_Voc_parents
,xin_Voc_parm
,xin_Voc_pascal
,xin_Voc_pascal16
,xin_Voc_password
,xin_Voc_path
,xin_Voc_pending
,xin_Voc_pentium
,xin_Voc_pic
,xin_Voc_picture
,xin_Voc_plitdli
,xin_Voc_plitest
,xin_Voc_pointer
,xin_Voc_pos
,xin_Voc_position
,xin_Voc_pp
,xin_Voc_pptrace
,xin_Voc_prec
,xin_Voc_precision
,xin_Voc_prefix
,xin_Voc_preproc
,xin_Voc_preview
,xin_Voc_print
,xin_Voc_priority
,xin_Voc_private
,xin_Voc_pro
```

Figure 122. Declare for the voc kind (Part 11 of 15)

```
                    ,xin_Voc_probe
                    ,xin_Voc_proc
                    ,xin_Voc_procedure
                    ,xin_Voc_proceed
                    ,xin_Voc_process
                    ,xin_Voc_profile
                    ,xin_Voc_protected
                    ,xin_Voc_ptr
                    ,xin_Voc_public
                    ,xin_Voc_put
                    ,xin_Voc_r
                    ,xin_Voc_range
                    ,xin_Voc_read
                    ,xin_Voc_real
                    ,xin_Voc_record
                    ,xin_Voc_recsize
                    ,xin_Voc_recursive
                    ,xin_Voc_red
                    ,xin_Voc_reducible
                    ,xin_Voc_reentrant
                    ,xin_Voc_refer
                    ,xin_Voc_refine
                    ,xin_Voc_regional
                    ,xin_Voc_relative
                    ,xin_Voc_release
                    ,xin_Voc_renames
                    ,xin_Voc_reorder
                    ,xin_Voc_repeat
                    ,xin_Voc_reply
                    ,xin_Voc_reread
                    ,xin_Voc_reserve
                    ,xin_Voc_reserved
                    ,xin_Voc_reserves
                    ,xin_Voc_resignal
                    ,xin_Voc_retcode
                    ,xin_Voc_return
                    ,xin_Voc_returns
                    ,xin_Voc_reuse
                    ,xin_Voc_revert
                    ,xin_Voc_rewrite
                    ,xin_Voc_rules
                    ,xin_Voc_runops
                    ,xin_Voc_s
                    ,xin_Voc_s386
                    ,xin_Voc_s486
                    ,xin_Voc_saa
                    ,xin_Voc_saa2
                    ,xin_Voc_saa3
                    ,xin_Voc_scalarvarying
                    ,xin_Voc_scheduler
```

Figure 122. Declare for the voc kind (Part 12 of 15)

```
                 ,xin_Voc_segmented
                 ,xin_Voc_select
                 ,xin_Voc_sem
                 ,xin_Voc_semantic
                 ,xin_Voc_seq
                 ,xin_Voc_seql
                 ,xin_Voc_sequence
                 ,xin_Voc_sequential
                 ,xin_Voc_set
                 ,xin_Voc_short
                 ,xin_Voc_signal
                 ,xin_Voc_signed
                 ,xin_Voc_single
                 ,xin_Voc_sis
                 ,xin_Voc_size
                 ,xin_Voc_sizefrom
                 ,xin_Voc_sizeto
                 ,xin_Voc_skip
                 ,xin_Voc_smessage
                 ,xin_Voc_smsg
                 ,xin_Voc_snap
                 ,xin_Voc_source
                 ,xin_Voc_spill
                 ,xin_Voc_sprog
                 ,xin_Voc_sql
                 ,xin_Voc_static
                 ,xin_Voc_stdcall
                 ,xin_Voc_stg
                 ,xin_Voc_stmt
                 ,xin_Voc_stop
                 ,xin_Voc_storage
                 ,xin_Voc_stream
                 ,xin_Voc_strg
                 ,xin_Voc_string
                 ,xin_Voc_stringrange
                 ,xin_Voc_stringsize
                 ,xin_Voc_struct
                 ,xin_Voc_structure
                 ,xin_Voc_strz
                 ,xin_Voc_subrg
                 ,xin_Voc_subscriptrange
                 ,xin_Voc_suspend
                 ,xin_Voc_sym
                 ,xin_Voc_syn
                 ,xin_Voc_syntax
                 ,xin_Voc_sysin
                 ,xin_Voc_sysparm
                 ,xin_Voc_sysprint
                 ,xin_Voc_system
                 ,xin_Voc_sz
```

Figure 122. Declare for the voc kind (Part 13 of 15)

```
                    ,xin_Voc_task
                    ,xin_Voc_term
                    ,xin_Voc_terminal
                    ,xin_Voc_test
                    ,xin_Voc_then
                    ,xin_Voc_thread
                    ,xin_Voc_tiled
                    ,xin_Voc_time
                    ,xin_Voc_title
                    ,xin_Voc_to
                    ,xin_Voc_total
                    ,xin_Voc_tp
                    ,xin_Voc_trace
                    ,xin_Voc_transient
                    ,xin_Voc_transmit
                    ,xin_Voc_trkofl
                    ,xin_Voc_tso
                    ,xin_Voc_tstack
                    ,xin_Voc_type
                    ,xin_Voc_u
                    ,xin_Voc_uen
                    ,xin_Voc_ufl
                    ,xin_Voc_unal
                    ,xin_Voc_unaligned
                    ,xin_Voc_unbuf
                    ,xin_Voc_unbuffered
                    ,xin_Voc_undefinedfile
                    ,xin_Voc_underflow
                    ,xin_Voc_undf
                    ,xin_Voc_union
                    ,xin_Voc_unlimited
                    ,xin_Voc_unlock
                    ,xin_Voc_unroll
                    ,xin_Voc_unsigned
                    ,xin_Voc_until
                    ,xin_Voc_update
                    ,xin_Voc_upperinc
                    ,xin_Voc_upthru
                    ,xin_Voc_v
                    ,xin_Voc_v1
                    ,xin_Voc_v2
                    ,xin_Voc_value
                    ,xin_Voc_valuelist
                    ,xin_Voc_valuerange
                    ,xin_Voc_var
                    ,xin_Voc_variable
                    ,xin_Voc_varying
                    ,xin_Voc_varyingz
                    ,xin_Voc_varz
                    ,xin_Voc_vb
```

Figure 122. Declare for the voc kind (Part 14 of 15)

```
                   ,xin_Voc_vbs
                   ,xin_Voc_virtual
                   ,xin_Voc_vs
                   ,xin_Voc_vsam
                   ,xin_Voc_w
                   ,xin_Voc_wait
                   ,xin_Voc_wcode
                   ,xin_Voc_when
                   ,xin_Voc_while
                   ,xin_Voc_windows
                   ,xin_Voc_winproc
                   ,xin_Voc_wkeep
                   ,xin_Voc_write
                   ,xin_Voc_x
                   ,xin_Voc_xchar
                   ,xin_Voc_xinfo
                   ,xin_Voc_xoptions
                   ,xin_Voc_xref
                   ,xin_Voc_zdiv
                   ,xin_Voc_zerodivide
               ) unsigned prec(16);
```

Figure 122. Declare for the voc kind (Part 15 of 15)

Notices

This information was developed for products and services offered in the U.S.A.
IBM may not offer the products, services, or features discussed in this document in
other countries. Consult your local IBM representative for information on the
products and services currently available in your area. Any reference to an IBM
product, program, or service is not intended to state or imply that only that IBM
product, program, or service may be used. Any functionally equivalent product,
program, or service that does not infringe any IBM intellectual property right may
be used instead. However, it is the user⌐s responsibility to evaluate and verify the
operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter
described in this document. The furnishing of this document does not give you
any license to these patents. You can send license inquiries, in writing, to:

> IBM Corporation
> J74/G4
> 555 Bailey Avenue
> San Jose, CA 95141-1099
> U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM
Intellectual Property Department in your country or send inquiries, in writing, to:

> IBM World Trade Asia Corporation
> Licensing
> 2-31 Roppongi 3-chome, Minato-ku
> Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

> INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors.

Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM	IMS
The IBM logo	IMS/ESA
ibm.com	Language Environment
AIX	MVS
CICS	OS/390
CICS/ESA	RACF
DB2	System/390
DFSMS	VisualAge
DFSORT	z/OS

Intel is a registered trademark of Intel Corporation in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and other countries.

Pentium is a registered trademark of Intel Corporation in the United States and other countries.

Unicode is a trademark of the Unicode Consortium.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be the trademarks or service marks of others.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Bibliography

Enterprise PL/I publications

Programming Guide, SC27-1457
Language Reference, SC27-1460
Messages and Codes, SC27-1461
Compiler and Run-Time Migration Guide, GC27-1458

PL/I for MVS & VM

Installation and Customization under MVS, SC26-3119
Language Reference, SC26-3114
Compile-Time Messages and Codes, SC26-3229
Diagnosis Guide, SC26-3149
Migration Guide, SC26-3118
Programming Guide, SC26-3113
Reference Summary, SX26-3821

z/OS Language Environment

Concepts Guide, SA22-7567

Debugging Guide, GA22-7560

Run-Time Messages, SA22-7566

Customization, SA22-7564

Programming Guide, SA22-7561

Programming Reference, SA22-7562

Run-Time Application Migration Guide, GA22-7565

Writing Interlanguage Communication Applications, SA22-7563

CICS Transaction Server

Application Programming Guide, SC33-1687

Application Programming Reference, SC33-1688

Customization Guide, SC33-1683

External Interfaces Guide, SC33-1944

DB2 UDB for z/OS

Administration Guide, SC26-9931

An Introduction to DB2 for z/OS, SC26-9937

Application Programming and SQL Guide, SC26-9933

Command Reference, SC26-9934

Messages and Codes, GC26-9940

SQL Reference, SC26-9944

DFSORT™

Application Programming Guide, SC33-4035

Installation and Customization, SC33-4034

IMS/ESA®

Application Programming: Database Manager, SC26-8015

Application Programming: Database Manager Summary, SC26-8037

Application Programming: Design Guide, SC26-8016

Application Programming: Transaction Manager, SC26-8017

Application Programming: Transaction Manager Summary, SC26-8038

Application Programming: EXEC DL/I Commands for CICS and IMS™, SC26-8018

Application Programming: EXEC DL/I Commands for CICS and IMS Summary, SC26-8036

z/OS MVS

JCL Reference, SA22-7597

JCL User's Guide, SA22-7598

System Commands, SA22-7627

z/OS UNIX System Services

z/OS UNIX System Services Command Reference, SA22-7802

z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA22-7803

z/OS UNIX System Services User's Guide, SA22-7801

z/OS TSO/E

Command Reference, SA22-7782

User's Guide, SA22-7794

z/Architecture

Principles of Operation, SA22-7832

Unicode® and character representation

z/OS Support for Unicode: Using Conversion Services, SC33-7050

Glossary

This glossary defines terms for all platforms and releases of PL/I. It might contain terms that this manual does not use. If you do not find the terms for which you are looking, see the index in this manual or *IBM Dictionary of Computing*, SC20-1699.

A

access. To reference or retrieve data.

action specification. In an ON statement, the ON-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever the appropriate condition is raised.

activate (a block). To initiate the execution of a block. A procedure block is activated when it is invoked. A begin-block is activated when it is encountered in the normal flow of control, including a branch. A package cannot be activated.

activate (a preprocessor variable or preprocessor entry point). To make a macro facility identifier eligible for replacement in subsequent source code. The %ACTIVATE statement activates preprocessor variables or preprocessor entry points.

active. The state of a block after activation and before termination. The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text. The state in which an event variable is said to be during the time it is associated with an asynchronous operation. The state in which a task variable is said to be when its associated task is attached. The state in which a task is said to be before it has been terminated.

actual origin (AO). The location of the first item in the array or structure.

additive attribute. A file description attribute for which there are no defaults, and which, if required, must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute*.

adjustable extent. The bound (of an array), the length (of a string), or the size (of an area) that might be different for different generations of the associated variable. Adjustable extents are specified as expressions or asterisks (or by REFER options for based variables), which are evaluated separately for each generation. They cannot be used for static variables.

aggregate. See *data aggregate*.

aggregate expression. An array, structure, or union expression.

aggregate type. For any item of data, the specification whether it is structure, union, or array.

allocated variable. A variable with which main storage is associated and not freed.

allocation. The reservation of main storage for a variable. A generation of an allocated variable. The association of a PL/I file with a system data set, device, or file.

alignment. The storing of data items in relation to certain machine-dependent boundaries (for example, a fullword or halfword boundary).

alphabetic character. Any of the characters A through Z of the English alphabet and the alphabetic extenders #, $, and @ (which can have a different graphic representation in different countries).

alphameric character. An alphabetic character or a digit.

alternative attribute. A file description attribute that is chosen from a group of attributes. If none is specified, a default is assumed. Contrast with *additive attribute*.

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

improving application performance 269
INCAFTER compiler option 31
INCDIR compiler option 32
include preprocessor
    syntax 88
INCLUDE statement
    compiler 77
INDEXAREA option 178
indexed data sets
    indexed sequential data set 173
indexed ESDS (entry-sequenced data set)
    DIRECT file 250
    loading 248
    SEQUENTIAL file 249
indicator variables, SQL 112
information interchange codes 171
INITFILL compiler suboption 23
initial volume label 174
initialization procedure of compiler user
  exit 388
INLINE compiler suboption 21
input
    data for PLISRTA 300
    data for sort 295
    defining data sets for stream
      files 195
    redirecting 187
    routines for sort program 295
    SEQUENTIAL 214
    skeletal code for sort 299
    sort data set 295
input/output
    compiler
        data sets 136
    data for compile and bind 119
    data for compile, prelink, and
      link-edit 122
    in cataloged procedures 118
    OS/390, punctuating long lines 144
    skeletal code for sort 297
    sort data set 294
INSOURCE option 34
Inter Language Communication
    linkage considerations 315
    with C 307
        data types 307
        Enum data type 308
        File type 309
        Functions returning ENTRYs 314
        parameter matching 310
        sharing output 317
        string parameter type
          matching 313
        structure data type 308
        using C functions 309
interactive program
    attention interrupt 34
interblock gap (IBG) 171
interchange codes 171
INTERNAL attribute 81
INTERRUPT compiler option 34
interrupts
    attention interrupts under interactive
      system 34
    ATTENTION ON-units 380
    debugging tool 380
    main description 379

invoking
    cataloged procedure 128
        link-editing multitasking
          programs 130
        multiple invocations 128

J

Java 320, 321, 323, 325, 326, 327, 329,
  330, 331, 335
JAVA 319
Java code, compiling 321, 325, 329, 331
Java code, writing 320, 323, 327, 331
JCL (job control language)
    improving efficiency 117
    using during compilation 135
jni
    JNI sample program 320, 323, 327,
      331

K

key indexed VSAM data set 236
key-sequenced data sets
    accessing with a DIRECT file 250
    accessing with a SEQUENTIAL
      file 249
    loading 248
    statements and options for 246
KEYLEN subparameter 175
KEYLENGTH option 178, 185
KEYLOC option
    usage 178
keys
    alternate index
        nonunique 253
        unique 253
    REGIONAL(1) data set 224
        dummy records 224
    VSAM
        indexed data set 236
        relative byte address 236
        relative record number 236
KEYTO option
    under VSAM 244
KSDS (key-sequenced data set)
    define and load 248
    unique key alternate index path 254
    updating 250
    VSAM
        DIRECT file 250
        loading 248
        SEQUENTIAL file 249
KSDS with VSAM
    indexed ESDS with VSAM
        access data set 249
        accessing data set 250
        updating data set 252

L

label
    for data sets 174
LANGLVL compiler option 35
Language Environment library xiv

large object (LOB) support, SQL
  preprocessor 108
LEAVE and REREAD options
    ENVIRONMENT option
        for consecutive data sets 213
length of record
    specifying under z/OS UNIX 168
library
    compiled object modules 192
    creating a data set library 190
    creating a member 193
    creating and updating a library
      member 191
    creating, examples of 191
    directory 190
    extracting information from a library
      directory 193
    general description of 174
    how to use 189
    information required to create 190
    placing a load module 192
    source statement 138
    source statement library 133
    SPACE parameter 190
    structure 193
    system procedure
      (SYS1.PROCLIB) 189
    types of 189
    updating a library member 193
    using 189
LIMCT subparameter 175, 230
LIMITS compiler option 36
line
    length 203
    numbers in messages 30
line feed (LF)
    definition 170
LINE option 196, 203
LINECOUNT compiler option 37
LINEDIR compiler option 37
LINESIZE option
    for tab control table 205
    OPEN statement 197
link-editing
    description of 141
LINKAGE compiler suboption
    effect on performance 274
    syntax 22
linkage considerations with ILC 315
LIST compiler option 37
listing
    cataloged procedure 128
    compiler
        aggregate length table 81
        ATTRIBUTE and cross-reference
          table 80
        ddname list 3
        file reference table 85
        heading information 79
        messages 85
        options 80
        preprocessor input 80
        return codes 85
        SOURCE option program 80
        statement nesting level 80
        statement offset addresses 82
        storage offset listing 84

Readers' Comments — We'd Like to Hear from You

Enterprise PL/I for z/OS
Programming Guide
Version 3 Release 7

Publication No. SC27-1457-07

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: 1-800-426-7773

If you would like a response from IBM, please fill in the following information:

_____   _____
Name   Address

_____   _____
Company or Organization

_____   _____
Phone No.   E-mail address

Fold and Tape   **Please do not staple**   Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL   PERMIT NO. 40   ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department HHX/H3
555 Bailey Ave.
San Jose, CA
 95141-1099

Fold and Tape   **Please do not staple**   Fold and Tape

IBM®

Program Number: 5655-H31

Printed in USA