VisualAge PL/I

IBM

Programming Guide

Version 2.1

GC26-9177-01

VisualAge PL/I

IBM

Programming Guide

Version 2.1

GC26-9177-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 523.

Second Edition (November 2002)

This edition applies to Version 2.1 of VisualAge PL/I Enterprise, part number 04L7217; and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

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

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

IBM Corporation, BWE/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

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

Contents

Part 1. Introducing PL/I on your workstation

Chapter 1. About this book

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

If you have typically used mainframe PL/I and are interested in moving your programs to the OS/2 or Windows platform, Chapter 2, “Porting applications between platforms” on page 7 should be particularly useful. Other information in this guide will help you understand some basic OS/2 and Windows features as well as give instructions on how to compile, link, and run a PL/I program.

Since OS/2 and Windows are different in some respects and since this guide is designed to accompany products on both platforms, differences between the two products are labeled in the text.

OS/2 This symbol marks the beginning of a section of text that applies only to OS/2. The end of each section is also marked.

WIN This symbol marks the beginning of a section of text that applies only to Windows.

In cases where an entire chapter is specific to either OS/2 or Windows, a note appears in the heading for that chapter. Material that is not marked should be common to both products.

What's new?

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

- Millennium Language Extensions designed to help you step up to the year 2000 challenge
- Some tips on how to use PL/I and Java together
- How to use dclgen in the Windows NT environment
- Help with using the Open Database Connectivity
- A chapter on using remote IMS DL/I support.

How to read the syntax diagrams

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

Arrow symbols

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

►►— Indicates the beginning of a statement.

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

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

—►◄ Indicates the end of a statement.

How to read syntax diagrams

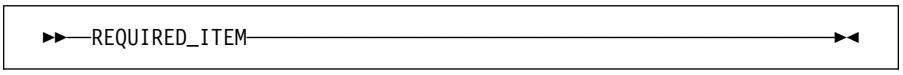
Diagrams of syntactical units other than complete statements start with the ►— symbol and end with the —► symbol.

Conventions

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

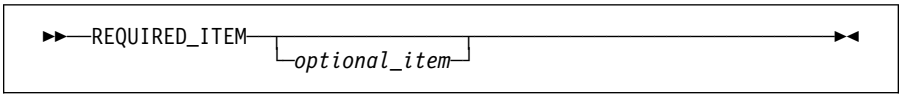
Required items

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

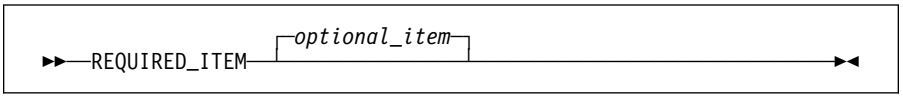


Optional Items

Optional items appear below the main path.

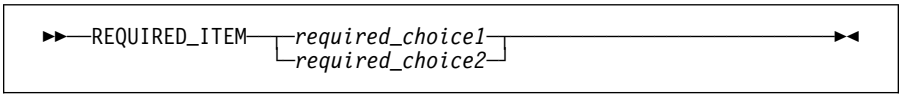


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

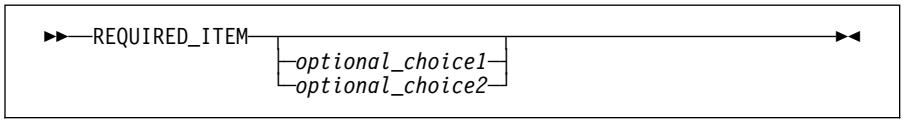


Multiple required or optional items

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

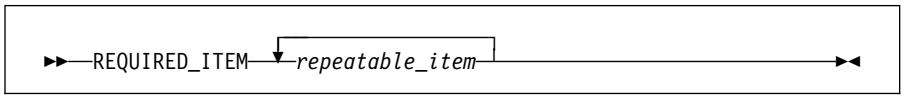


If choosing one of the items is optional, the entire stack appears below the main path.

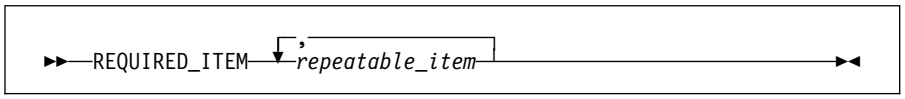


Repeatable items

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



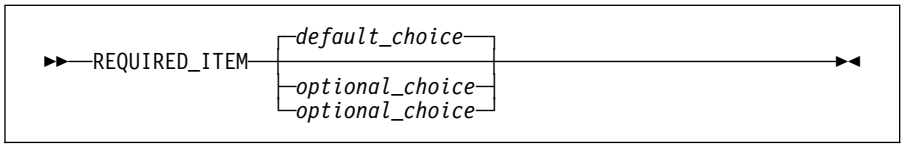
If the repeat arrow contains a comma, you must separate repeated items with a comma.



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

Default keywords

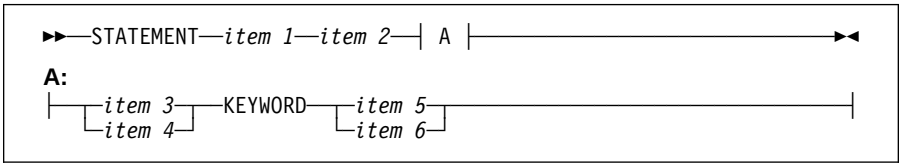
Default keywords appear above the main path, and the remaining choices are shown below the main path.



Fragments

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

How to read syntax diagrams



Chapter 2. Porting applications between platforms

Getting mainframe applications to compile on the workstation

The IBM mainframe environment has a different hardware and operating system architecture than your AIX system or your personal computer (PC). Operating systems other than the mainframe are sometimes referred to as *workstation* platforms. In this book, we use the term workstation to refer to the AIX, OS/2, and Windows operating systems collectively.

Because of fundamental platform differences as well as difference the OS PL/I compiler and the VisualAge PL/I compilers, some problems can arise as you move PL/I programs between the mainframe and workstation environments. This chapter describes some of these differences between development platforms, and then provides instructions that minimize problems in the following areas:

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

Getting mainframe applications to compile on the workstation

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

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

Embedded control characters

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

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

National characters and other symbols

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

$
#
@

To avoid potential problems involving “not,” “or” and the currency symbol, use the NOT (see “NOT” on page 64), OR (see “OR” on page 67), and CURRENCY (see “CURRENCY” on page 42) compile-time options on the *PROCESS statement. Avoid potential problems involving other characters by using the NAMES (see

"NAMES" on page 63) compile-time option to define extralingual characters and symbols.

Choosing the right compile-time options

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

Most compile-time options are set to be compatible with OS PL/I, but a few, such as LANGLVL(SAA), are not. If you want to improve compatibility with OS PL/I, you could specify the following options:

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

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

These (and all the other compiler) options are listed alphabetically in Chapter 4, "Compile-time option descriptions" on page 35 where they are also described in detail.

Understanding workstation limitations of OS PL/I language features

The following section describes some OS PL/I language features that the VisualAge PL/I compiler either does not support at all or supports but with some restrictions.

Multitasking language

VisualAge PL/I does not support tasking language.

DEFINED attribute

The VisualAge PL/I compiler supports the DEFINED attribute except:

- while it supports string-overlay defining and simple defining, it does not support iSUB defining
- it does not support the use of DEFINED variables in PUT DATA or GET DATA statements if the DEFINED variables:
  - have the attributes BIT or GRAPHIC, or
  - are defined on an element of an array.

Array expressions

Array expressions are supported by VisualAge PL/I but with some restrictions. In particular, an array expression may be used as:

- the source in a multiple assignment.
- an argument to a user function with the restriction that if the array expression has a string data type, then it must have known, constant length
- an argument to the SUM or PROD built-in functions (and these functions will always be done inline)

- an argument to the ALL or ANY built-in functions with the restriction that that the array expression must be either a NONVARYING BIT array reference or an array expression with known, constant length

Structure expressions
Structure expressions are supported by VisualAge PL/I but with some restrictions. In particular:

- In an assignment statement (with or without BY NAME),
  - a structure expression may be used as the source if there is only one target
  - but only a structure reference may be used as the source if there are multiple targets
- Structure expressions may be used in PUT LIST and PUT EDIT.
- In an invocation of a user function,
  - if a parameter to the function is not described, then only an unparenthesized structure reference may be used as an argument
  - if a parameter to the function is described, then
    - if the parameter has constant extents, then any structure expression with matching levels may be used as an argument
    - if the parameter has nonconstant extents, then only an unparenthezided structure reference with matching levels and data attributes may be used as an argument

pseudovariables
The VisualAge PL/I compiler does not support the COMPLEX pseudovariable.

The other OS PL/I pseudovariables are supported with the exception that a pseudovariable can appear as the target in a DO-specification only if it is the SUBSTR, REAL, IMAG or UNSPEC psuedovariable.

POLY built-in function
The POLY built-in function is supported but with the restrictions that

- The first argument must be REAL FLOAT
- The second argument must be scalar

UNLOCK statement
The UNLOCK statement is not supported

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

Using the macro facility to help port programs

In many cases, potential portability problems can be avoided by using the macro facility because it has the capability of isolating platform-specific code. For example, you can

include platform-specific code in a compilation for a given platform and exclude it from compilation for a different platform.

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

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

```
%dcl compiletime builtin;

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

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

Getting mainframe applications to run on the workstation

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

Linking differences

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

Every .DLL that you build must have at least one module compiled with the DLLINIT compile-time option (see “DLLINIT” on page 50).

Data representations causing run-time differences

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

The workstation compilers support options that instruct the operating system to treat data and floating-point operations the same way that the mainframe does. There are

suboptions of the DEFAULT compile-time option that you should specify for all mainframe applications that might need to be changed when moving code to the workstation:

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

For more information on these compile-time options, see “DEFAULT” on page 43.

ASCII vs. EBCDIC

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

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

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

While 'a' < 'A' is true for EBCDIC, it is false for ASCII.

While 'A' < '1' is true for EBCDIC, it is false for ASCII.

While x >= '0' almost always means that x is a digit in EBCDIC, this is not true for ASCII.

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

For information on converting from ASCII to EBCDIC, see “Using data conversion tables” on page 384.

NATIVE vs. NONNATIVE

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

be held as '02010000'x on OS/2 or Windows and as '00000102'x on AIX or the mainframe.

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

The OS/2 and Windows representation is known, conversely, as Little Endian (Little End In)

This difference in internal representations affects:

- FIXED BIN variables requiring two or more bytes
- OFFSET variables
- The length prefix of VARYING strings
- Ordinal and area data

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

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

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

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

a1 = a2;
```

IEEE vs. HEXADEC

Workstation operating systems represent floating-point data using the IEEE format while the mainframe traditionally uses the hexadecimal format.

Table 1 on page 14 summarizes the differences between normalized floating-point IEEE and hexadecimal:

Getting mainframe applications to run on the workstation

Table 1. Normalized IEEE vs. normalized hexadecimal

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

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

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

Also, while FIXED BIN calculations produce the same result independent of the internal representations described above, floating-point calculations do not necessarily produce the same result because of the differences in how the floating-point values are represented. This is particularly true for short and extended floating-point.

EBCDIC DBCS vs ASCII DBCS

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

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

Environment differences affecting portability

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

File names

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

```
d:\programs\data\myfile.dat
```

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

File attributes

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

Control codes

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

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

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

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

dcl f file output;

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

write file(f) from(a);
```

Device-dependent control codes

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

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

Language elements causing run-time differences

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

Getting mainframe applications to run on the workstation

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

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

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

INITIAL attribute for AREAs is ignored

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

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

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

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

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

a(1) = a1;
a(2) = a2;
a(3) = a3;
a(4) = a4;
```

Issuing of ERROR messages

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

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

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

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

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

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

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

Enablement of OVERFLOW and ZERODIVIDE

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

- OVERFLOW or ZERODIVIDE is raised and the corresponding ON-unit is entered.
- Control does not leave the ON-unit through a GOTO statement.

Getting mainframe applications to run on the workstation

Part 2. Compiling and linking your program

Chapter 3. Compiling your program

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

A short practice exercise

Try compiling, linking, and running a simple program to get an idea of how to use PL/I in the OS/2 or Windows environment.

The HELLO program

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

1. **Create the source program**
   Create a file, HELLO.PLI, with the following PL/I statements.

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

   Leave the first space of every line blank: by default, the compiler only recognizes characters in columns 2-72. (For additional information, see “MARGINS” on page 60.)

   Save the file to disk.

2. **Compile the program**
   In a window or full-screen session, go to the directory that contains the HELLO.PLI file and enter the following command:

   ```
   pli hello
   ```

   The compiler displays information about the compilation on your screen, and creates the object file (HELLO.OBJ) in the current directory.

3. **Link the program**
   Without changing directories, enter the following command:

   ```
   ilink hello.obj
   ```

   This combines the file HELLO.OBJ with needed library files (as specified by the LIBS compile-time option), producing the file HELLO.EXE (the executable program) in the same directory.

   Since no parameters are specified with the link command, the defaults are used. (The options available with the link command are described in Chapter 7, “Linking your program” on page 132.)

4. **Run the program**
   Without changing directories, enter the following command:

   ```
   hello
   ```

   This invokes the HELLO.EXE program, which displays `Hello!` on your monitor.

Preparing to compile source programs

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

Using compile-time options

As you prepare to compile programs, consider using a subset of the available compile-time options. For a complete description of the compile-time options, including their optional abbreviated forms, see Chapter 4, “Compile-time option descriptions” on page 35.

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

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

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

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

Using the sample programs provided with the product

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

OS/2 For OS/2, the sample programs are installed in the \IBMPLI\SAMPLES directory. A readme file `smoread.me`. is provided for the sample programs.

WIN For Windows, the sample programs are installed in the \Program Files\IBM\VAPLI\SAMPLES directory. A readme file `smwread.me`. is provided for the sample programs.

Preparing to compile source programs

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

Program file structure

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

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

If your program requires %PROCESS or *PROCESS statements, they must be the first lines in your source file. The first line after them that does not consist entirely of blanks

or comments must be a PACKAGE or PROCEDURE statement. The last line of your source file that does not consist entirely of blanks or comments must be an END statement matching the PACKAGE or PROCEDURE statement.

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

Using a PROCEDURE statement with PROCESS:

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

 /*  optional comments */

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

Using a PACKAGE statement with PROCESS:

```
*PROCESS  ;
*PROCESS  ;
*PROCESS  ;

 /*  optional comments */

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

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

INCLUDE processing

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

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

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

```
%include member;
%include ddname(member);
```

The compiler searches for this file in the following order:

1. The directories specified in the environment variable IBM.DDNAME, if the %include statement specified a *ddname*
2. The directories specified in the environment variable IBM.SYSLIB
3. The directories specified in the environment variable INCLUDE
4. The current directory.

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

%OPTION directive

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

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

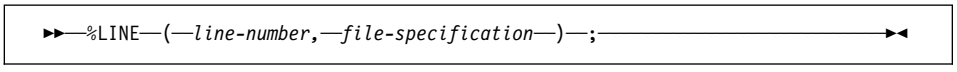
The compile-time options or directives that can be used with the %OPTION directive include:

- LANGLVL(SAA)
- LANGLVL(SAA2)
- When dealing with messages and information generated for debugging, you can use LINE(*line-number*,'*file-specification*') to indicate that the next line should be treated as if it came from the specified line and file. The *line-number* must be an integral value and the *file-specification* must be in quotes. For an example of these lines, compile a program using the options PPTRACE, MACRO, and MDECK.

See Chapter 4, “Compile-time option descriptions” on page 35 for option descriptions.

%LINE directive

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



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

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

Margins

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

You can change the default margin setting (see “MARGINS” on page 60). If you choose to keep the default setting, your source code should begin in column 2.

Note: The %PROCESS (or *PROCESS) statement is an exception to the margin rule and *must* start in the first column. For more information about the %PROCESS statement, see “%PROCESS statement” on page 33.

Program file format

The compiler, running under the OS/2 and Windows operating systems, expects the contents of your source file to consist of ASCII format and CR-LF type[1]. If you created your file on a workstation, the format should be correct; however, if you transfer a file from another machine environment, make sure that the file transfer utility does any needed translation (to ASCII and CR-LF).

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

Line continuation

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

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

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

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

Preparing to compile source programs

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

```
do;
 if x > 200 then
  display ('This is a long string and requires more than one line to
   type it into my program');
 else
  display ('This is a short string');
end;
```

You should use the following sequence of statements:

```
do;
 if x > 200 then
  display ('This is a long string and requires more than '
   ||'one line to type it into my program');
 else
  display ('This is a short string');
end;
```

Compile-time options summary

Before you compile your program, review the following summary of compile-time option functions so you can select the most useful options. Each option is also listed (alphabetically) and described in more detail in Chapter 4, “Compile-time option descriptions” on page 35. For information about how to use compile-time options to optimize your code, see Chapter 19, “Improving performance” on page 359.

Table 2 (Page 1 of 2). What compile-time options can do for you

Objective	Option	Function	Default
Control how your source program input is interpreted by the compiler	ADDEXT	Determines if extensions are added to file names	ADDEXT
	BLANK	Specifies extralingual characters allowed as "blanks"	See page 39
	GRAPHIC	Recognizes DBCS data	NOGRAPHIC
	INCAFTER	Specifies a file to be included	See page 53
	INCLUDE	Specifies %INCLUDE file-name extensions	See page 53
	MARGINS	Specifies margins for source code	MARGINS(2,72)
	NAMES	Specifies extralingual characters allowed in identifiers	See page 63
	NOT	Specifies logical NOT symbols (^)	NOT('^')
	OR	Specifies logical OR symbols (|)	OR('|')
	WINDOW	Sets the value for the window argument	WINDOW(1950)
Control enforcement of PL/I language rules	DEFAULT	Specifies defaults for data attributes/OPTIONS	See page 43
	LANGLVL	Specifies language level	See page 55
	LIMITS	Specifies maximum lengths	See page 57
	RULES	Specifies certain language rules	See page 73
	RESPECT	Honors the DATE attribute specification	RESPECT()
	USAGE	Specifies certain language rules	See page 82
Use the macro facility	INSOURCE	Lists source input to macro facility	NOINSOURCE
	MACRO	Invokes macro facility	NOMACRO
	MDECK	Saves modified source from the macro facility	NOMDECK
	SYSPARM	Specifies returned value of SYSPARM macro facility built-in function	SYSPARM('')
Control the listing	AGGREGATE	Lists aggregates and their lengths	NOAGGREGATE
	ATTRIBUTES	Lists attributes	NOATTRIBUTES
	FLAG	Flags messages of a specified severity level	FLAG(W)
	LINECOUNT	Specifies number of lines per listing page	LINECOUNT(60)
	LIST	Lists machine (assembler-like) instructions	NOLIST
	MARGINI	Shows source margins in listing	MARGINI(' ')
	MAXMSG	Specifies maximum number of messages allowed	MAXMSG(W 250)
	NATLANG	Specifies national language for messages	NATLANG(ENU)
	NEST	Shows statement nesting levels	NONEST
	OPTIONS	Lists compile-time options in effect	NOOPTIONS
	SOURCE	Lists source statements	NOSOURCE
	TERMINAL	Displays diagnostic messages on the screen	TERMINAL
	XINFO	Controls the generation of extra information	See page 84
	XREF	Lists where each identifier is referenced and where each is set	NOXREF

Setting compile-time environment variables

Table 2 (Page 2 of 2). What compile-time options can do for you

Objective	Option	Function	Default
Prevent subsequent processing stages	COMPILE	Controls the code generation stage	NOCOMPILE
	EXIT	Enables the compiler user exit	NOEXIT
	PP	Controls selection of preprocessors	NOPP
	PROCEED	Controls continuation of preprocessing	NOPROCEED
	SEMANTIC	Controls semantic stage	NOSEMANTIC
	SYNTAX	Controls syntax stage	NOSYNTAX
Control object code generation	CHECK	Tells compiler to monitor ALLOCATE/FREE	CHECK(NOSTORAGE)
	CMPAT	Determines format of descriptors	CMPAT(LE)
	DLLINIT	Define object files for use in .EXE's or .DLL's	NODLLINIT
	INTERRUPT	Makes object code capable of responding to attention interrupts	NOINTERRUPT
	LIBS	Names default libraries to be searched at link time	LIBS
	OBJECT	Generates object module	OBJECT
	PROBE	Controls generation of stack probes	PROBE
	SYSTEM	Specifies target execution environment	See page 80
	WIDECHAR	Specifies byte-order for WIDECHAR data	See page 83
Facilitate program testing	GONUMBER	Includes statement numbers	NOGONUMBER
	PREFIX	Enables or disables PL/I conditions for the compilation unit	See page 69
	SNAP	Specifies whether SNAP and PLIDUMP traceback output is complete	NOSNAP
	TEST	Specifies capabilities used during testing	NOTEST
Optimize your code	IMPRECISE	Controls the precision of floating-point results and interrupt location	NOIMPRECISE
	OPTIMIZE	Produces optimized code	NOOPTIMIZE
	REDUCE	Reduces some structure assignments	REDUCE

Setting compile-time environment variables

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

OS/2 On OS/2, environment variables can be set either from the command line, in a command (`.cmd`) file, or in the CONFIG.SYS file. If it is set on the command line or by running a command file, the options are only in effect for the current session (until you reboot your computer). If it is set in the CONFIG.SYS file, the options are set when you boot your computer and stay in effect unless you override them using a `.cmd` file or by specifying options on the command line.

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

Setting compile-time environment variables

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

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

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

The default location for compiler input and output is the current directory; the IBM-supplied default for each compile-time option is specified in Chapter 4, "Compile-time option descriptions" on page 35.

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

IBM.OPTIONS

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

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

The syntax of the character string you assign to the IBM.OPTIONS environment variable is the same as that required for the compile-time options specified on the PLI command (see "Using the PLI command to invoke the compiler" on page 32).

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

IBM.PPINCLUDE

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

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

The syntax of the character string you assign to the IBM.PPINCLUDE environment variable is the same as that required for the compile-time options specified on the PLI command (see "Using the PLI command to invoke the compiler" on page 32).

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

Setting compile-time environment variables

IBM.PPMACRO

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

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

The syntax of the character string you assign to the IBM.PPMACRO environment variable is the same as that required for the compile-time options specified on the PLI command (see “Using the PLI command to invoke the compiler” on page 32).

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

IBM.PPSQL

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

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

The syntax of the character string you assign to the IBM.PPSQL environment variable is the same as that required for the compile-time options specified on the PLI command (see “Using the PLI command to invoke the compiler” on page 32).

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

IBM.PPCICS

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

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

The syntax of the character string you assign to the IBM.PPCICS environment variable is the same as that required for the compile-time options specified on the PLI command (see “Using the PLI command to invoke the compiler” on page 32).

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

IBM.SOURCE

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

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

Note: If you specify a path with the OS/2 file specification on the PLI command, the compiler ignores the IBM.SOURCE environment variable.

IBM.SYSLIB

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

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

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

IBM.PRINT

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

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

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

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

IBM.OBJECT

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

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

An object file contains the machine code translation of your PL/I source statements. To make it executable, you must link it with any other OBJ files that comprise your program, and with appropriate library files. For a summary of how to link your program, see Chapter 7, “Linking your program” on page 132.

IBM.DECK

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

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

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

INCLUDE

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

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

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

TMP

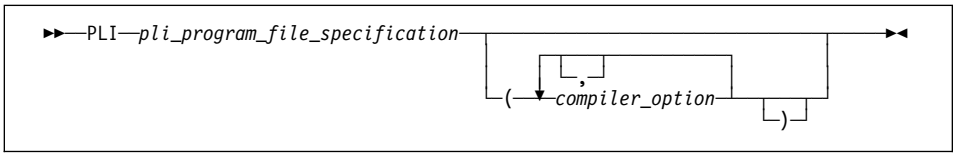
The TMP environment variable specifies the input and output directory for any temporary work files that the compiler needs. For example:

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

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

Using the PLI command to invoke the compiler

Use the PLI command to invoke the compiler. You can enter it on the OS/2 command line or in a CMD file.



pli_program_file_specification
The OS/2 or Windows file specification for your primary source program file. If you omit the extension from your file specification, the compiler assumes an extension of PLI. If you omit the complete path, the current directory is assumed, unless you specify otherwise using IBM.SOURCE.

compiler_option
One or more compile-time options, described in Chapter 4, “Compile-time option descriptions” on page 35.

The following is an example of the PLI command:

```
pli hello (source
```

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

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

When using response files, remember these guidelines:

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

Where to specify compile-time options

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

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

IBM.OPTIONS and IBM.PPxxx environment variables

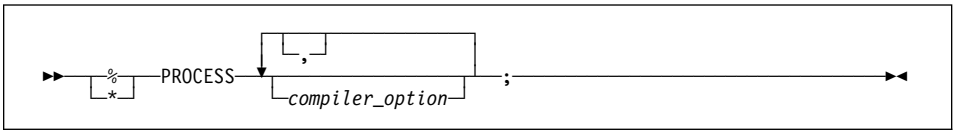
The first way you can specify options is to set the IBM.OPTIONS environment variable for compile-time options and IBM.PPxxx environent variables for preprocessor options. See “Setting compile-time environment variables” on page 28. Controlling compile-time options with these environment variables overrides the normal option defaults.

PLI command

The second way to specify compile-time options—overriding option defaults and IBM.OPTIONS and IBM.PPxxx—is on the PLI command when you invoke the compiler (see “Using the PLI command to invoke the compiler” on page 32). The options apply only to the current compilation.

%PROCESS statement

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



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

```
%process source margins(1,80);
 Hello: proc options(main);
        display('Hello!');
 end Hello;
```

You can specify one or more %PROCESS statements, but they must precede all other PL/I source statements, including blank lines.

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

split a keyword or value. It is recommended that, instead of wrapping the statement, you code multiple %PROCESS statements, one per line.

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

Chapter 4. Compile-time option descriptions

Compile-time options

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

Rules for using compile-time options

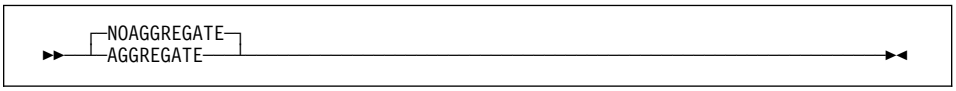
1. If you specify mutually exclusive compile-time options or suboptions, the last one you specify takes effect.
2. If required strings conform to PL/I identifier rules, you do not need to enclose them in quotes. The compiler folds these strings to uppercase.

   The following options should have their string specifications enclosed in quotes, because the string specifies either special characters or run-time options:

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

AGGREGATE

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



Abbreviations: NAG, AG

The Aggregate Length 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 *.

Default: NOAGGREGATE

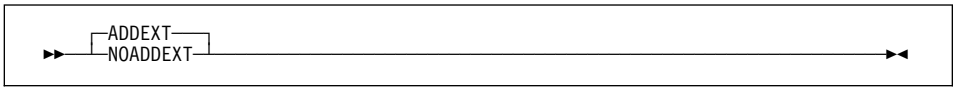
A sample listing is shown in “Using the compiler listing” on page 123.

Examples:

```
aggregate
noaggregate
```

ADDEXT

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



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

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

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

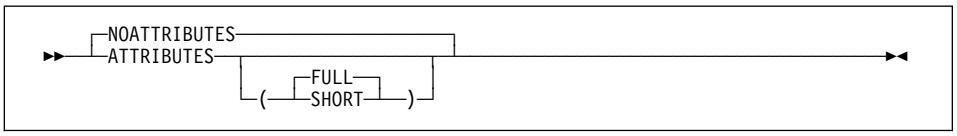
Default: ADDEXT

Examples:

```
addext
noaddext
```

ATTRIBUTES

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



Abbreviations: NA, A

FULL
List all identifiers and attributes. For an example of the table produced when you select ATTRIBUTES(FULL), see "Using the compiler listing" on page 123.

SHORT
Omit unreferenced identifiers.

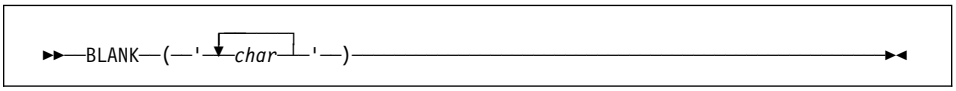
Default: NOATTRIBUTES

Examples:

```
attributes(full)
noattributes
```

BLANK

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



Note: Do not code any blanks between the quotes.

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

char
A single SBCS character.

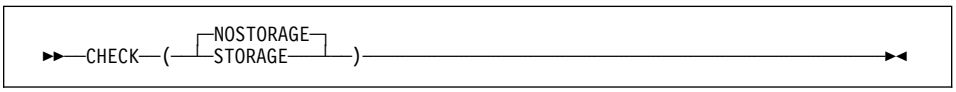
You cannot specify any of the alphabetic characters, digits, and special characters defined in the *PL/I language reference information*.

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

Default: BLANK('09'x)

CHECK

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



Abbreviations: STG, NSTG

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

- ALLOCSIZE
- CHECKSTG

- UNALLOCATED

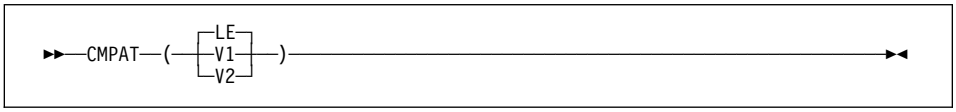
Default: CHECK(NOSTORAGE)

Examples:

```
check(stg)
check(nostorage)
```

CMPAT

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



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

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

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

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

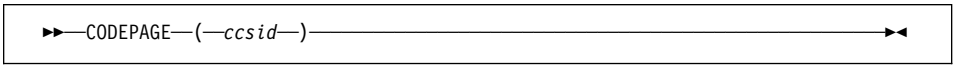
The DFT(DESCLIST) option conflicts with the CMPAT(V1) or CMPAT(V2) option, and if it is specified with either the CMPAT(V1) or the CMPAT(V2) option, a message will be issued and the DFT(DESCLOCATOR) option assumed.

Default: CMPAT(LE)

CODEPAGE

The CODEPAGE option specifies the code page used for:

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



The supported CCSID's are:

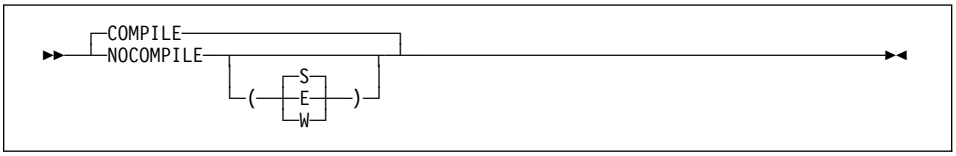
01047	01145	00273	00297
01140	01146	00277	00500
01141	01147	00278	00871
01142	01148	00280	00819
01143	01149	00284	00813
01144	00037	00285	00920

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

Default: CODEPAGE(00819)

COMPILE

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



Abbreviations: NC, C

NOCOMPILE
Compilation halts unconditionally after semantic checking.
NOCOMPILE(S)
Compilation halts if a severe or unrecoverable error is detected.
NOCOMPILE(E)
Compilation halts if an error, severe error, or unrecoverable error is detected.
NOCOMPILE(W)
Compilation halts if a warning, error, severe error, or unrecoverable error is detected.
COMPILE
Equivalent to NOCOMPILE(S).

If the compilation is terminated by the NOCOMPILE option, whether or not listings are

produced depends on when the compilation stopped. For example, cross-reference and attribute listings should be produced with the NOCOMPILE option, but an error might occur during semantic checking that stops those listings from being produced.

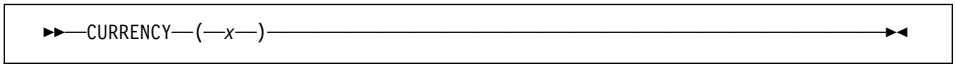
Default: NOCOMPILE(S)

Examples:

```
compile
nocompile(s)
```

CURRENCY

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



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

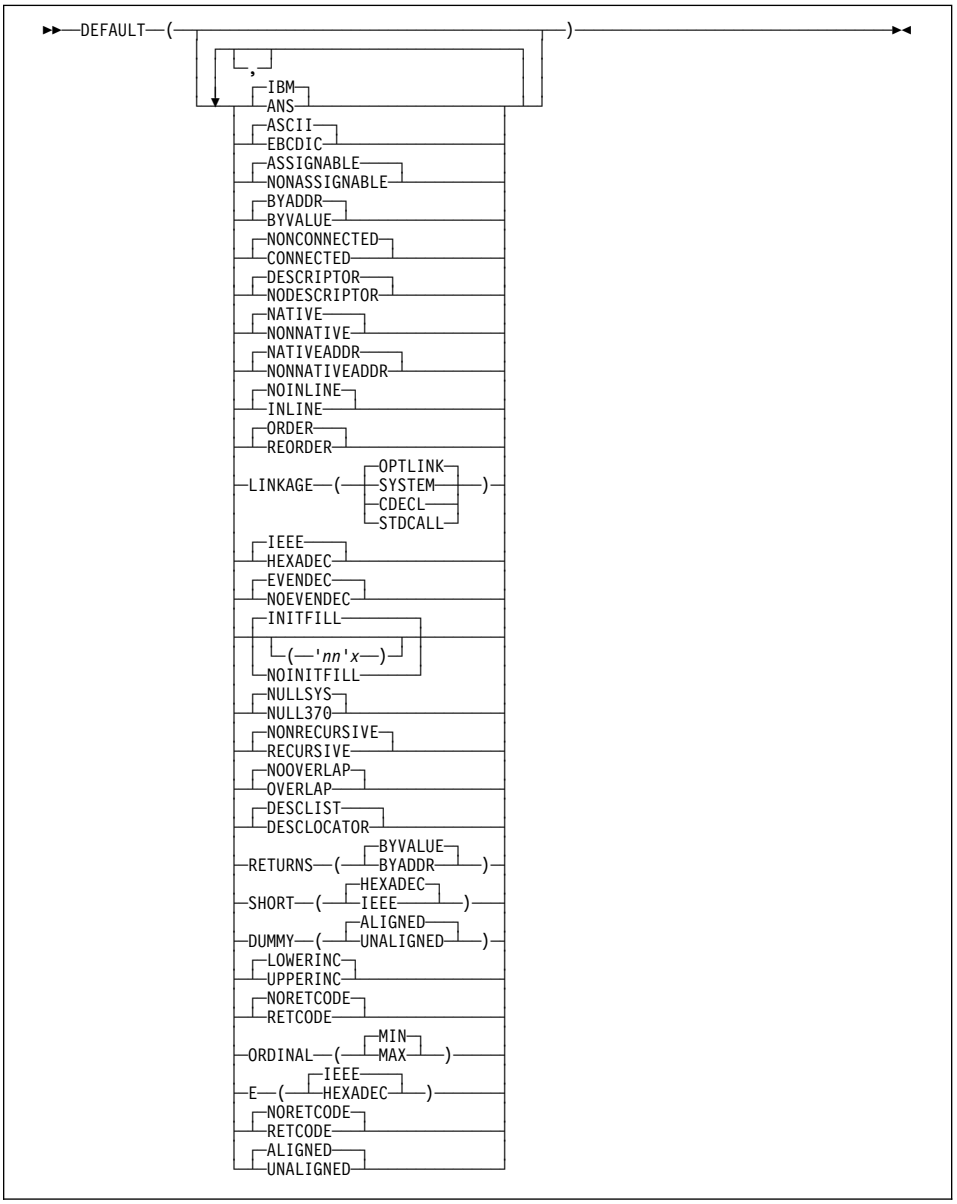
Default: CURRENCY('$')

Example:

```
currency('#')
```

DEFAULT

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



Abbreviations: DFT, ASGN, NONASGN, CONN, NONCONN

DEFAULT

IBM or ANS

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

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

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

ASCII or EBCDIC

The ASCII and EBCDIC suboption is not supported by the Personal Edition of PL/I workstation products.

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

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

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

```
'123'A is the same as '313133'X
'123'E is the same as 'F1F1F3'X
```

ASSIGNABLE or NONASSIGNABLE

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

BYADDR or BYVALUE

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

CONNECTED or NONCONNECTED

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

DEFAULT

DESCRIPTOR or NODESCRIPTOR

Using DESCRIPTOR with a PROCEDURE indicates that a descriptor list was passed, while DESCRIPTOR with ENTRY indicates that a descriptor list should be passed. NODESCRIPTOR results in more efficient code, but yields errors under the following conditions:

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

NATIVE or NONNATIVE

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

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

If your program bases fixed binary variables on pointer or offset variables (or conversely, pointer or offset variables on fixed binary variables), specify either:

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

Other combinations produce unpredictable results.

NATIVEADDR or NONNATIVEADDR

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

If your program bases fixed binary variables on pointer or offset variables (or conversely, pointer or offset variables on fixed binary variables), specify either:

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

Other combinations produce unpredictable results.

INLINE or NOINLINE

This option sets the default for the inline procedure option.

Specifying INLINE allows your code to run faster but, in some cases, also creates a larger executable file. For more information on how inlining can improve the

DEFAULT

performance of your application, see Chapter 19, “Improving performance” on page 359.

ORDER or REORDER
Affects optimization of the source code. Specifying REORDER allows optimization of your source code, see

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

OVERLAP or NOOVERLAP
If you specify OVERLAP, the compiler presumes the source and target in an assignment can overlap and generates, as needed, extra code in order to ensure that the result of the assignment is okay. Chapter 19, “Improving performance” on page 359.

LINKAGE
The linkage convention for procedure invocations is:

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

SYSTEM
The standard linking convention for OS/2 APIs. All parameters are passed on the stack, but the *calling* function cleans up the stack.

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

CDECL
This linkage convention is available on both OS/2 and Windows. All parameters are passed on the stack, but the *calling* function cleans up the stack. External names have _ applied as a prefix.

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

For more detailed information on linkage conventions, see Chapter 24, “Calling conventions” on page 420.

IEEE or HEXADEC
This suboption is not supported by the Personal Edition of PL/I workstation products.

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

EVENDEC or NOEVENDEC

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

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

If you specify EVENDEC and then assign 123 to a FIXED DEC(2) variable, the SIZE condition is raised. If you specify NOEVENDEC, the SIZE condition is not raised (just as it would not be raised of you were using mainframe PL/I).

EVENDEC is the default.

INITFILL or NOINITFILL

This suboption controls the default initialization of automatic variables.

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

NOINITIFILL is the default.

LOWERINC or UPPERINC

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

LOWERINC is the default.

NULLSYS or NULL370

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

NULLSYS is the default.

RECURSIVE or NONRECURSIVE

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

NONRECURSIVE is the default.

DESCLIST or DESCLOCATOR

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

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

DEFAULT

DESCLIST is the default.

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

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

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

SHORT (HEXADEC) is the default.

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

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

Consider the following example:

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

dcl x entry( fixed bin(31) );

call x( b3 );
```

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

DUMMY(ALIGNED) is the default.

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

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

ALIGNED or UNALIGNED

This suboption allows you to force byte-alignment on all of your variables.

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

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

ALIGNED is the default.

E(IEEE or HEXADEC)

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

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

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

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

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

IEEE is the default.

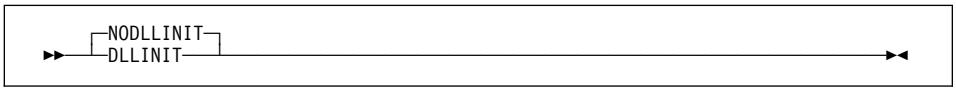
Default: DEFAULT (IBM ASCII ASSIGNABLE BYADDR NONCONNECTED DESCRIPTOR NATIVE NATIVEADDR NOINLINE ORDER LINKAGE(OPTLINK) IEEE EVENDEC NOINITFILL ORDINAL(MIN) NOOVERLAP NULLSYS NONRECURSIVE DESCLIST RETURNS(BYVALUE) SHORT(HEXADEC) DUMMY(ALIGNED) LOWERINC NORETCODE ALIGNED E(IEEE)).

Example:

```
dft ( byvalue ans linkage(system) )
```

DLLINIT

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



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

DLLINIT
This option must be specified in at least one of your compilations when you use the object files produced by those compilations to build a .DLL.

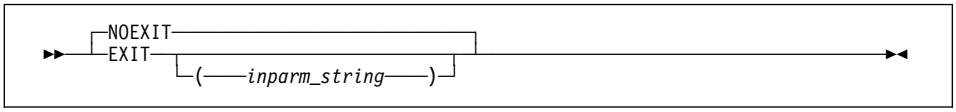
Default: NODLLINIT

Examples:

```
nodllinit
dllinit
```

EXIT

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



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

Default: NOEXIT

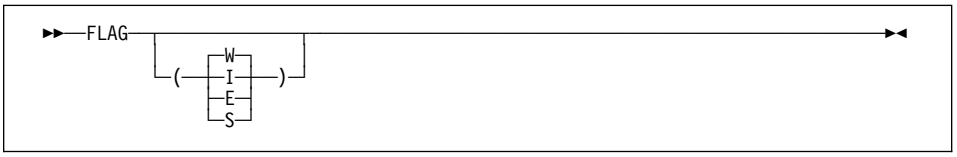
For more information, see Chapter 20, “Using user exits” on page 375.

Example:

```
exit ('Abcd')
```

FLAG

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



Abbreviation: F

I List all messages.

W List all except information messages.

E List all except warning and information messages.

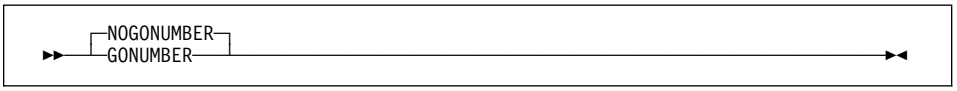
S List only severe error and unrecoverable error messages.

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

Default: FLAG(W)

GONUMBER

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



Abbreviations: NGN, GN

Default: NOGONUMBER

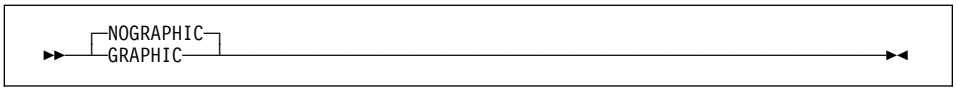
Examples:

```
nogonumber
gonumber
```

GRAPHIC •IMPRECISE

GRAPHIC

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



Abbreviations: NGR, GR

You must specify GRAPHIC if you use any of the following in your source program:

- DBCS identifiers
- DBCS in comments
- Graphic string constants
- Mixed string constants

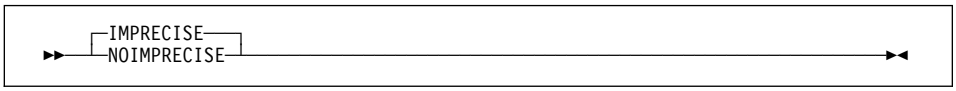
Default: NOGRAPHIC

Examples:

```
nographic
graphic
```

IMPRECISE

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



Abbreviations: IMP, NIMP

IMPRECISE

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

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

NOIMPRECISE

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

WIN Although NOIMPRECISE does provide better floating-point error detection than IMPRECISE, the Windows operating system does not allow immediate detection of floating-point exceptions. If you have a statement in your

program that is likely to raise a floating-point exception, you can avoid this detection problem by enclosing the statement, by itself, in a BEGIN block.

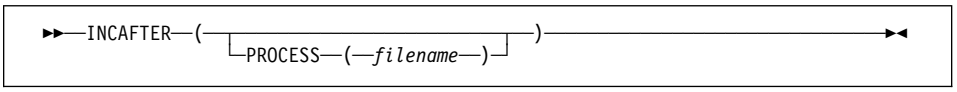
Default: IMPRECISE

Examples:

```
noimprecise
imprecise
```

INCAFTER

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



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

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

Consider the following example:

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

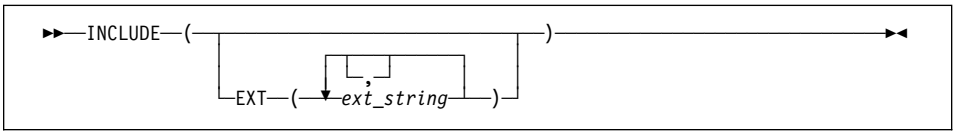
This example is equivalent to having the statement %INCLUDE DFTS; after the last PROCESS statement in your source.

Example:

```
incafter(process(dfts))
```

INCLUDE

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



INSOURCE

Abbreviation: INC

The extension string (see the note on *strings* in step 2 on page 37 under “Rules for using compile-time options”) can be up to 31 characters long, but it is truncated to the first three characters.

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

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

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

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

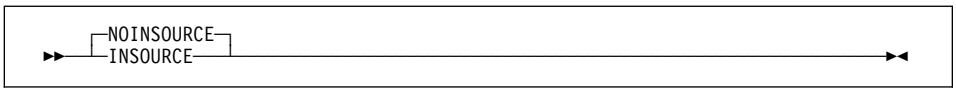
```
include ( ext(Cop Inc '2++' Mac) )
```

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

```
include (ext(' ',Inc,Cpy,Mac))
```

INSOURCE

The INSOURCE option specifies that the compiler should include a listing of the source program before the OS PL/I Version 2 macro preprocessor translates it.



Abbreviations: NIS, IS

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

FULL is the default.

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

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

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

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

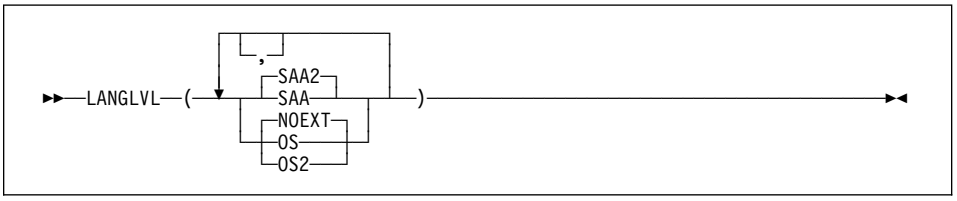
The INSOURCE listing will contain all of the main program before any of the included text from the file "member" (and it would contain all of that file before any text included by it - and so on).

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

Default: NOINSOURCE

LANGLVL

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



SAA

This suboption is not supported by the Personal Edition of PL/I workstation products.

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

SAA2

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

NOEXT

No extensions beyond the language level specified are allowed.

OS

The ENVIRONMENT options unique to the MVS and VM/CMS mainframe environments are allowed, such as Variable Unblocked (V), Variable Blocked (VB), Variable Spanned (VS), and Variable Blocked Spanned (VBS). For a complete list of options that are valid in MVS and VM/CMS, see the OS PL/I Programming Guide.

LIBS

OS2 (or Windows)
The ENVIRONMENT options unique to the OS/2 environment are allowed.

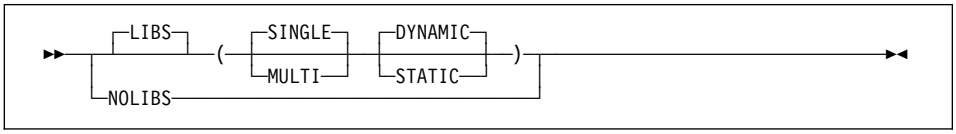
Default: LANGLVL(SAA2 NOEXT)

Example:

```
langlvl (saa2)
langlvl (saa os)
```

LIBS

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



LIBS
Same as specifying LIBS(SINGLE DYNAMIC)

LIBS(SINGLE DYNAMIC)
Specifies that default libraries searched at link time are the single-threading PL/I libraries:

- OS/2 On OS/2, these are `ibmos20i.lib`, `ibmostbi.lib`, `hepos20i.lib`, and `os2386.lib`.
- WIN On Windows, these are `ibmws20i.lib`, `ibmwstbi.lib`, `hepws20i.lib`, and `kernel32.lib`.

LIBS(MULTI DYNAMIC)
Specifies that default libraries searched at link time are the multi-threaded PL/I libraries:

- OS/2 On OS/2, these are `ibmom20i.lib`, `ibmomtbi.lib`, `hepom20i.lib`, and `os2386.lib`.
- WIN On Windows, these are `ibmwm20i.lib`, `ibmwmtbi.lib`, `hepwm20i.lib`, and `kernel32.lib`.

LIBS(SINGLE STATIC)
Specifies that default libraries searched at link time are the static, non-multithreading libraries:

- OS/2 On OS/2, these are `ibmos20.lib`, `ibmos30.lib`, `ibmostb.lib`, `hepos20.lib`, and `os2386.lib`.
- WIN On Windows, these are `ibmws20.lib`, `ibmws35.lib ibmwstb.lib`, `hepws20.lib`, and `kernel32.lib`.

LIBS(MULTI STATIC)
Specifies that default libraries searched at link time are the static, multi-threaded libraries. This means the library will be statically linkied into the user module.

- OS/2 On OS/2, these are `ibmom20.lib`, `ibmom30.lib`, `ibmomtb.lib`, `hepom20.lib`, and `os2386.lib`.
- WIN On Windows, these are `ibmwm20.lib`, `ibmwm35.lib ibmwmtb.lib`, `hepwm20.lib`, and `kernel32.lib`.

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

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

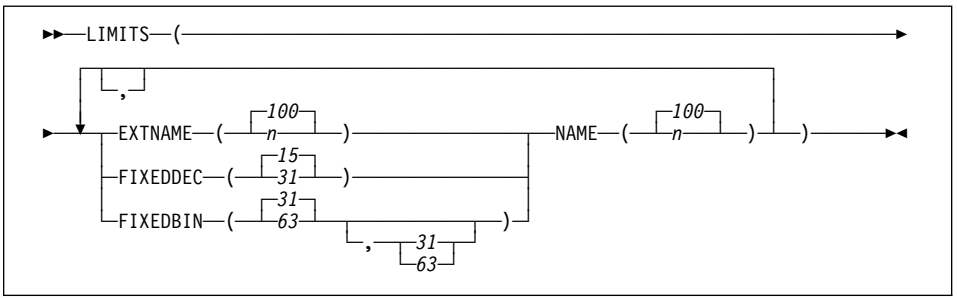
Default: LIBS

Examples:

```
libs(single static)
nolibs
```

LIMITS

This option specifies various implementation limits.



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

FIXEDDEC
Specifies the maximum precision for FIXED DECIMAL.

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

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

FIXEDBIN(63,31) is not allowed.

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

NAME
Specifies the maximum length of variable names in your program. The maximum value for *n* is 100; the minimum value is 7.

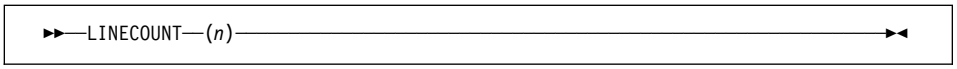
Default: LIMITS(EXTNAME(100) FIXEDBIN(31,31) FIXEDDEC(15) NAME(100))

Example:

```
limits(extname(8))
```

LINECOUNT

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



Abbreviations: LC

The value of *n* can be from 1 to 32,767.

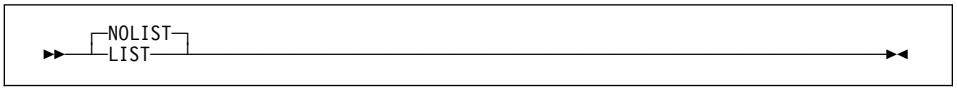
Default: LINECOUNT(60)

Example:

```
linecount(55)
```

LIST

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



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

Assembler listings do not always compile. A sample listing is shown in “Using the compiler listing” on page 123.

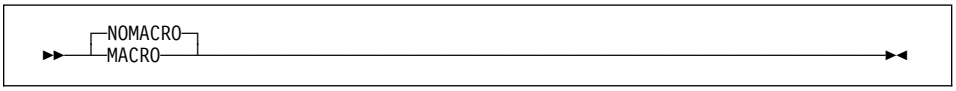
Default: NOLIST

Examples:

```
nolist
list
```

MACRO

The MACRO option causes the macro facility to be invoked prior to compilation. If both MACRO and PP(MACRO) are specified, the macro facility is invoked twice. When the MACRO option is used, MACRO('*macro-options*') is inserted into the PP option.



Abbreviations: NM, M

For example, if the following compile-time options are specified:

```
MDECK NOINSOURCE MACRO PP(MACRO SQL)
```

The PP option is modified and effectively becomes:

```
PP (MACRO MACRO SQL)
```

See also “PP” on page 68.

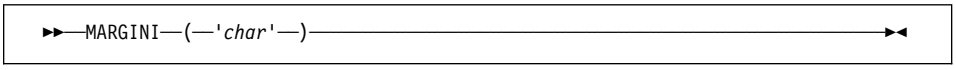
Default: NOMACRO

Example:

```
macro
```

MARGINI

This option specifies the margin indicator used in the source listing produced.



Abbreviations: MI('char')

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

Default: MARGINI(' ')

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

For a sample listing, see “Using the compiler listing” on page 123.

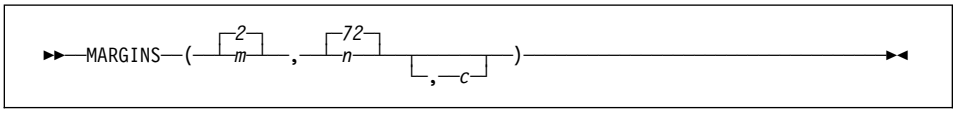
MARGINS

Example:

```
margini('*')
```

MARGINS

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



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.

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

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

(blank) Skip one line before printing

0 Skip two lines before printing

– Skip three lines before printing

\+ No skip before printing

1 Start new page

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

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

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

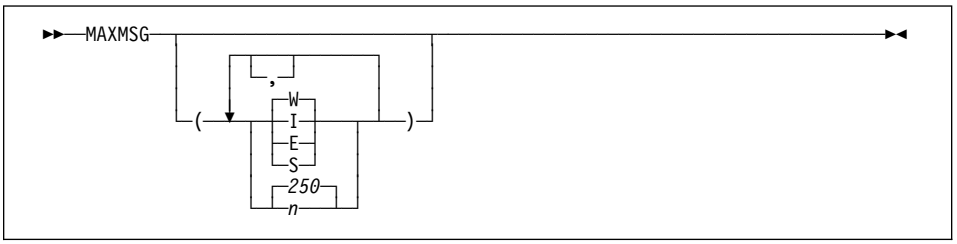
Default: MARGINS (2 72)

Example:

```
margins(5,72)
```

MAXMSG

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



I Count all messages.

W Count all except information messages.

E Count all except warning and information messages.

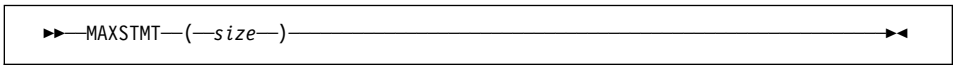
S Count only severe error and unrecoverable error messages.

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

Default: MAXMSG(W 250)

MAXSTMT

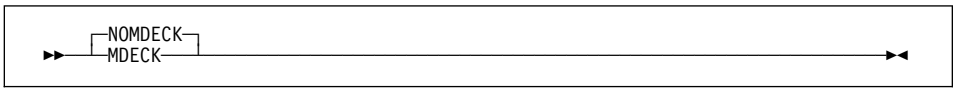
Under the MAXSTMT option, if the MSG(390) option is also in effect, the compiler will flag any block that has more than the specified number of statements. On Windows, however, optimization of such a block will not be turned off.



Default: MAXSTMT(4096)

MDECK

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



Abbreviations: NMD, MD

MDECK is ignored if NOMACRO is in effect. See “MACRO” on page 59 for an example.

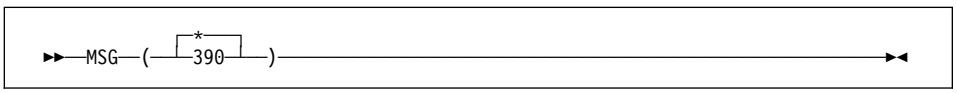
Default: NOMDECK

Examples:

```
nomdeck
mdeck
```

MSG

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



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

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

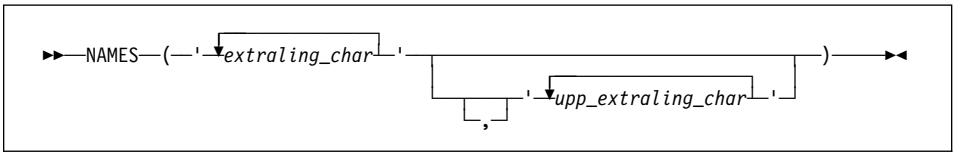
Default: MSG(*)

Examples:

```
msg(390)
```

NAMES

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



extraling_char
An extralingual character.

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

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

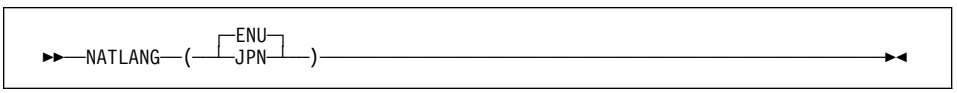
Default: NAMES('#@$' '#@$')

Example:

```
names('äöüß' 'ÄÖÜß')
```

NATLANG

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



ENU
US English, mixed case.

JPN
Japanese

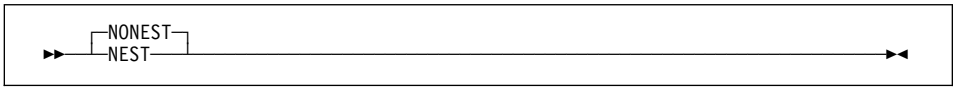
Default: NATLANG(ENU)

Example:

```
natlang (jpn)
```

NEST

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



For an example of the source listing, see “Using the compiler listing” on page 123.

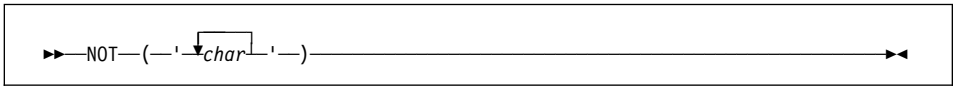
Default: NONEST

Examples:

```
nonest
nest
```

NOT

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



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

Default: NOT ('^')

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

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

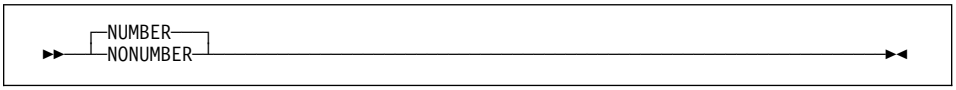
Example:

```
not('\}')
not('^\}')
```

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

NUMBER

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



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

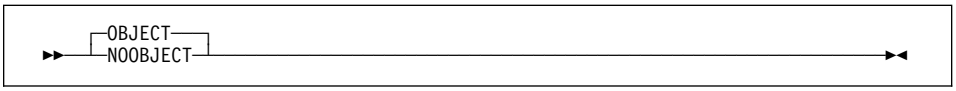
NUMBER and STMT are mutually exclusive. Specifying NONUMBER implies STMT.

Abbreviations: NUM, NNUM

Default: NUMBER

OBJECT

This option specifies whether object code is produced.



Abbreviations: OBJ, NOBJ

The module is saved in the current directory.

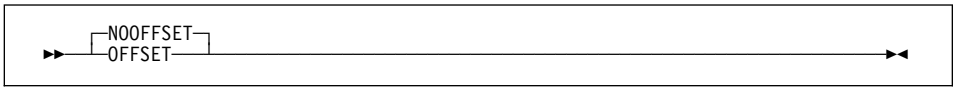
Default: OBJECT

Examples:

```
noobj
object
```

OFFSET

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



Abbreviations: OF, NOF

The `.cod` file contains the offset and machine code for every instruction generated. Use the sample program `cod2off` to reduce the size of this file to a listing of the offset for the start of each statement in every block of the compilation.

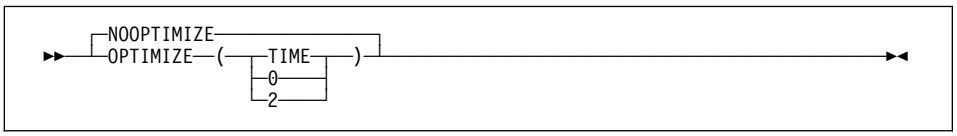
Default: NOOFFSET

Examples:

```
nooffset
offset
```

OPTIMIZE

This option specifies the type of optimization required.



Abbreviations: NOPT, OPT

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

OPTIMIZE(TIME) or OPTIMIZE(2)
Use either of these options to cause extended optimizations of the object code and produce faster running object code. Optimization requires additional compile time, but usually results in reduced run time.

Inlining occurs only under optimization.

See Chapter 19, “Improving performance” on page 359 for a full discussion of optimization.

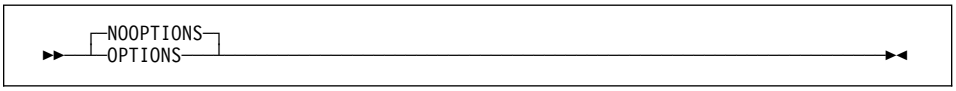
Default: NOOPTIMIZE

Examples:

```
opt(2)
nooptimize
```

OPTIONS

This option produces a listing of all compile-time options in effect for the compilation (see Chapter 6, “Compilation output” on page 122 for an example).



Abbreviations: NOP, OP

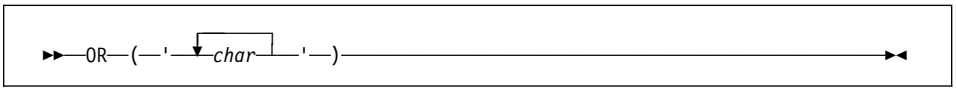
Default: NOOPTIONS

Examples:

```
nop
options
```

OR

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



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

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

Default: OR ('|')

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

Example:

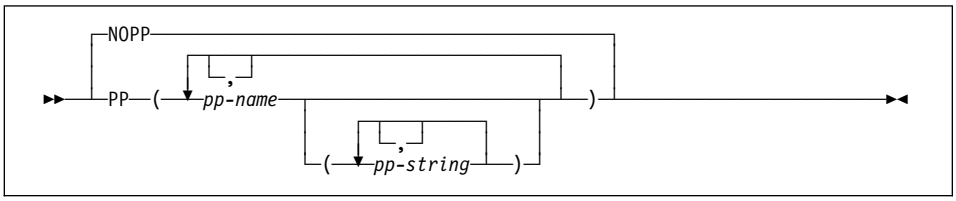
```
or('\}')
or('|\}')
```

PP

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

PP

This option specifies which (and in what order) preprocessors are invoked prior to compilation. The MACRO option and the PP(MACRO) option both cause the macro facility to be invoked prior to compilation. If both MACRO and PP(MACRO) are specified, the macro facility is invoked twice. The same preprocessor can be specified multiple times.



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

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

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

Default: NOPP

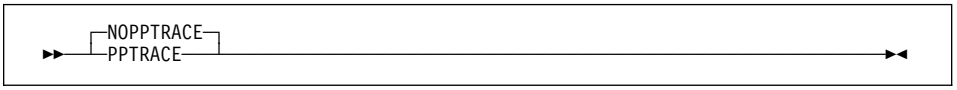
You can specify a maximum of 31 preprocessors.

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

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

PPTRACE

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



PPTRACE should be used only with preprocessors other than those that are integrated with the PL/I compiler.

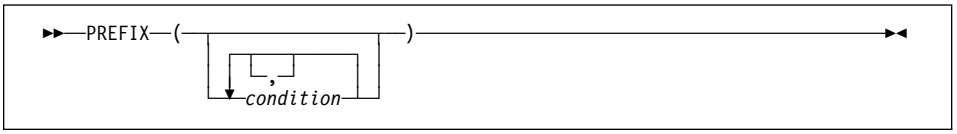
Default: NOPPTRACE

Examples:

```
pptrace
nopptrace
```

PREFIX

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



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

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

Example: Given the following source:

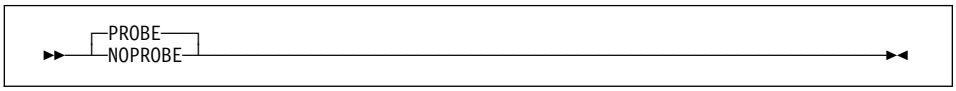
```
   (stringsize):
name: proc options (reentrant reorder);
end;
```

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

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

PROBE

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



PROBE
Specifies that stack probes are generated.

NOPROBE
No generation of stack probes.

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

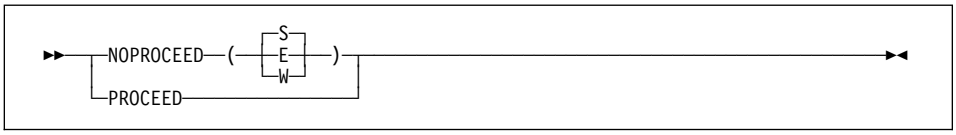
Default: PROBE

Examples:

```
probe
noprobe
```

PROCEED

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



Abbreviations: PRO, NPRO

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

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

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

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

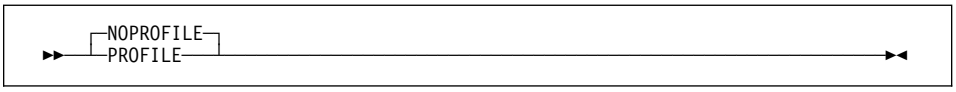
Default: NOPROCEED(S)

Examples:

```
npro(s)
proceed
```

PROFILE

This option specifies whether or not information for the profiler is generated.



Abbreviations: PROF, NPROF

NOPROFILE
Information for the profiler is not generated.

PROFILE
Information for the profiler is generated.

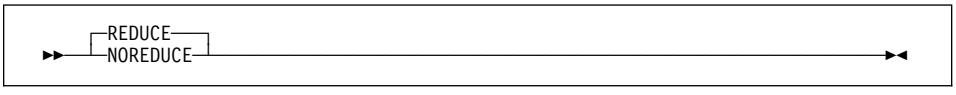
Default: NOPROFILE

Examples:

```
noprofile
prof
```

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.



RESPECT

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 less executable code to be generated for an assignment of a null string to a structure, and that will usually mean your code will run much faster. However, under the REDUCE option, any assignment of a null string to a structure that is reduced to a simple copy will also cause any padding bytes in that structure to be filled with '00'x.

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

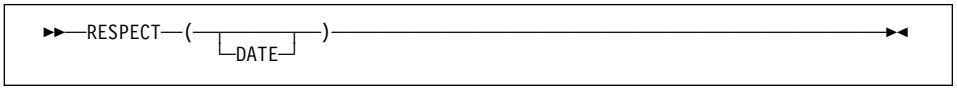
```
dcl
 1 struc,
   5  field10          bin fixed(31),
   5  field11          dec fixed(13)
   5  field12          bin fixed(15),
   5  field13          char(2);
```

Under the NOREDUCE option, the assignment *struc = '';* will cause four assignments to be generated, but the padding byte will be unchanged. However, under the REDUCE option, the assigment would be reduced to one simple copy (a MVC), but the padding byte will be set to a '00'x.

Default: REDUCE

RESPECT

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



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

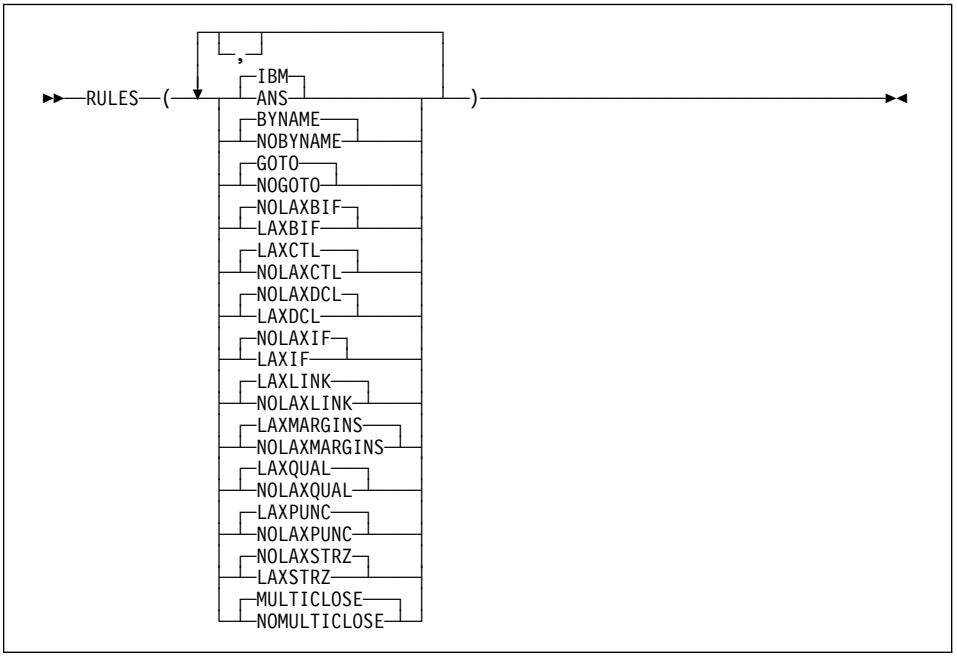
Default RESPECT()

Examples:

```
respect(date)
```

RULES

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



Abbreviations: LAXCOM, NOLAXCOM

IBM or ANS

Under the IBM suboption:

- For operations requiring string data, data with the BINARY attribute is converted to BIT.
- Conversions in arithmetic operations or comparisons occur as described in the *OS PL/I Language Reference*.
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *OS 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.

RULES

Under the ANS suboption:

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

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

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

Use NOGOTO to have all GOTO statements flagged.

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

NOLAXBIF is the default.

LAXCOMMENT or NOLAXCOMMENT
If you specify RULES(LAXCOMMENT), the compiler ignores the special characters /*/. Whatever comes between sets of these characters, then, is interpreted as part of the syntax rather than as a comment. If you specify RULES(NOLAXCOMMENT), the compiler treats /*/ as the start of a comment which continues until a closing */ is found.

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

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

LAXCTL or NOLAXCTL
Specifying LAXCTL allows a CONTROLLED variable to be declared with a constant extent and yet to be allocated with a differing extent. NOLAXCTL

requires that if a CONTROLLED variable is to be allocated with a varying extent, then that extent must be specified as an asterisk or as a non-constant expression.

The following is illegal under NOLAXCTL:

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

LAXDCL or NOLAXDCL

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

LAXIF or NOLAXIF

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

The following are illegal under NOLAXIF:

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

LAXLINK or NOLAXLINK

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

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

LAXMARGINS or NOLAXMARGINS

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

LAXPUNC | NOLAXPUNC

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

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

LAXQUAL or NOLAXQUAL

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

SEMANTIC

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

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

LAXSTRZ or NOLAXSTRZ
Specifying LAXSTRZ causes the compiler not to flag any bit or character variable that is initialized to or assigned a constant value that is too long if the excess bits are all zeros (or if the excess characters are all blank).

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

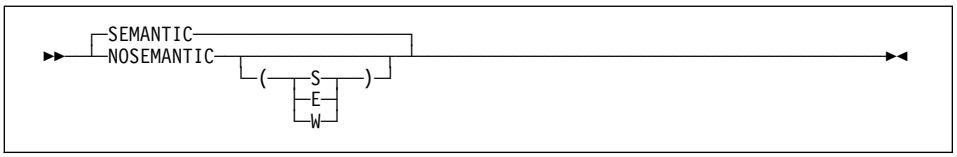
Default: RULES (IBM BYNAME GOTO NOLAXBIF NOLAXCOMMENT LAXCTL LAXDCL LAXIF LAXLINK LAXMARGINS LAXPUNC LAXQUAL NOLAXSTRZ MULTICLOSE)

Examples:

```
rules(ans)
rules(ans laxdcl laxif nolaxqual)
```

SEMANTIC

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



Abbreviations: SEM, NSEM

SEMANTIC
Equivalent to NOSEMANTIC(S).

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

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

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

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

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

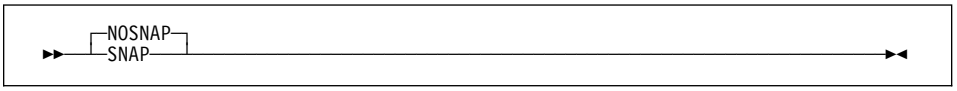
Default: NOSEMANTIC(S)

Examples:

```
nsem(w)
semantic
```

SNAP

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



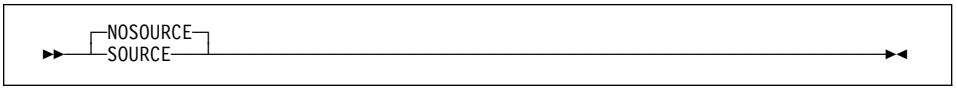
SNAP
SNAP and PLIDUMP traceback output always includes all PL/I routines on the call stack. SNAP can significantly increase the size and reduce the performance of your programs. It is not recommended for production programs.

NOSNAP
SNAP and PLIDUMP traceback output may not be complete.

Default: NOSNAP

SOURCE

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



Abbreviation: S, NS

SOURCE
The complier produces a listing of the source.

NOSOURCE
The compiler does not produce a source listing.

A source listing is not produced unless syntax checking is performed.

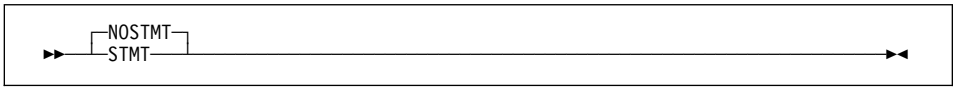
Default: NOSOURCE

Examples:

```
nosource
s
```

STMT

The STMT option specifies that statements in the source program are to be counted and that this "statement number" is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, SOURCE and XREF options.



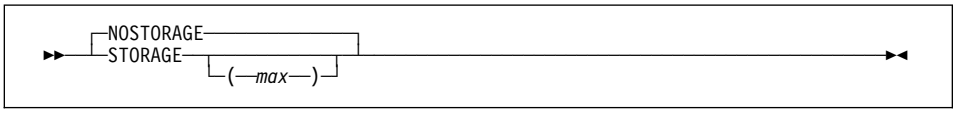
Specifying NOSTMT implies NUMBER.

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

Default: NOSTMT

STORAGE

This option determines whether or not the compiler produces a report in the listing which gives the approximate amount of stack storage used by each block in your program.



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*. If you specify STORAGE, but not specify a value, the default for *max* is 1000.

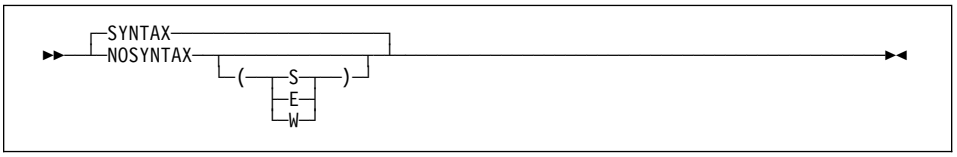
Default: NOSTORAGE

Examples:

```
storage(1400)
nostorage
```

SYNTAX

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



Abbreviations: SYN, NSYN

SYNTAX
Equivalent to NOSYNTAX(S).

NOSYNTAX
No syntax checking is performed.

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

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

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

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

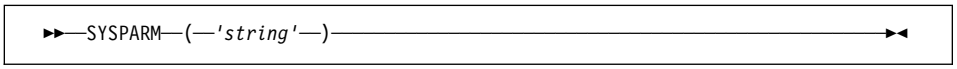
Default: NOSYNTAX(S)

Examples:

```
nosyntax(e)
syntax
```

SYSPARM

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



string
This string can be up to 64 characters long. A null string is the default, however, if you choose to specify a string value, see the note on *strings* in step 2 on page 37 under “Rules for using compile-time options.”

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

Default: SYSPARM('')

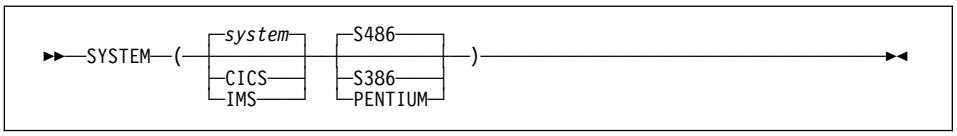
Example:

```
sysparm('debug')
```

SYSTEM

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

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



system
The value for this variable should be OS/2 or Windows depending on the system you are using.

CICS
Specifies that the program runs under CICS.

IMS
Specifies that the program will run under IMS.

S386
The object code is intended to run on a machine which has an 80386 or compatible chip. The code also runs on machines with 486 or Pentium chips.

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

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

Default: For OS/2, SYSTEM(OS2 S486); for Windows, SYSTEM(Windows S486)

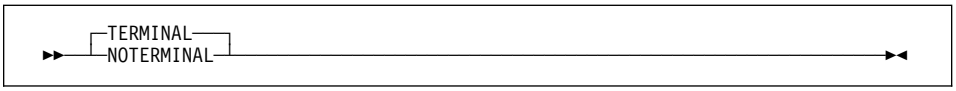
For MAIN procedures being compiled with SYSTEM(CICS) OPTIONS (BYVALUE) is assumed and PROCEDURE OPTIONS(BYADDR), if specified, is diagnosed.

Example:

```
system(os2 s486)
```

TERMINAL

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



Abbreviations: TERM, NTERM

TERMINAL
Messages are displayed on the terminal.

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

Default: TERMINAL

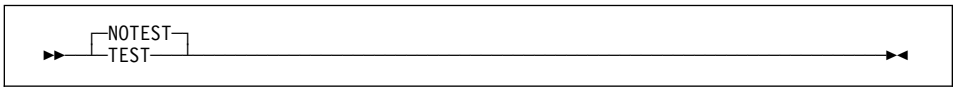
Example:

```
noterminal
```

TEST •USAGE

TEST

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



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

NOTEST
Suppresses the generation of all testing information.

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

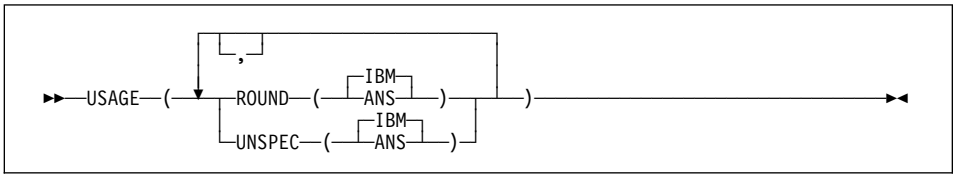
Default: NOTEST

Example:

```
test
```

USAGE

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



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

Under the ROUND(ANS) suboption, the ROUND built-in function is implemented as described in the *OS PL/I Version 2 Language Reference*.

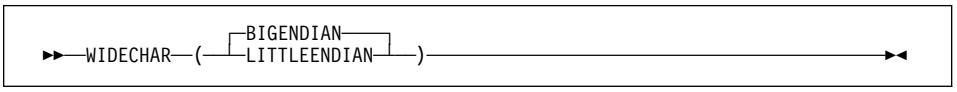
UNSPEC(IBM | ANS)
Under the UNSPEC(IBM) suboption, UNSPEC cannot be applied to a structure and, if applied to an array, returns an array of bit strings.

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

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

WIDECHAR

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



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

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

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

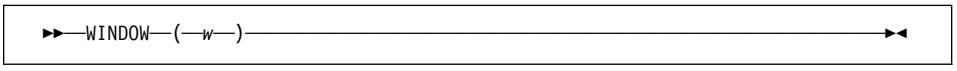
Default: WIDECHAR(LITTLEENDIAN)

Example:

```
WIDECHAR( BIGENDIAN )
```

WINDOW

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



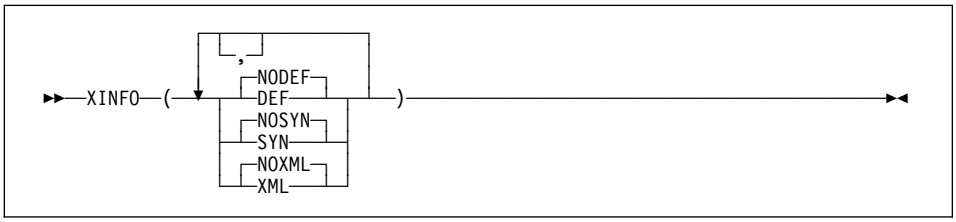
The value for *w* is either an unsigned integer that represents the start of a fixed window or a negative integer that specifies a "sliding" window. For example, Window(-20) indicates a window that starts 20 years prior to the year when the program runs.

Default: WINDOW(1950)

XINFO

XINFO

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



DEF

Causes the compiler to generate a .DEF file which you can later use as a template to build a DLL. The .DEF file lists all of the external names exported by the compilation unit. The names of external files and conditions defined in a package are not actually exported and do not appear in the generated .DEF file.

The .DEF has the same name as the .PLI file and is saved in the current directory unless you specify otherwise with the IBM.OBJECT environment variable.

NODEF

No .DEF file is generated.

SYN

Specifies that the compiler create a file to be used by the Year 2000 Analysis tool.

NOSYN

No file is produced.

XML

An XML side-file is created. This XML file includes:

- the file reference table for the compilation
- the block structure of the program compiled
- the messages produced during the compilation

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

The DTD file for the XML produced is:

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

<!ELEMENT compilation ((procedure)*,(message)*,FileReferenceTable)>
<!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 IncludedFromFile (#PCDATA)>
<!ELEMENT IncludedOnLine (#PCDATA)>
```

NOXML
No XML side-file is created.

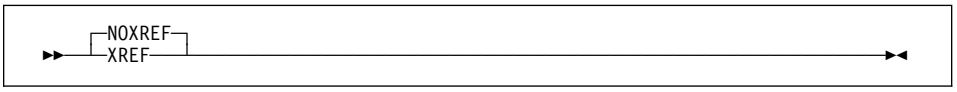
Default: XINFO(NOMSG NOSYM NOXML)

Examples:

```
xinfo(def)
xinfo(syn)
```

XREF

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



Abbreviations: X, NX

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

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

In addition to the cross-reference list, the compiler produces a listing of unreferenced identifiers. In this list, variables do not appear if they are named constants or static nonassignable variables. If any field in a union or structure is

referenced, the name of the union or structure does not appear. Level 1 names for unions or structures appear only if none of the members are referenced.

For an example and description of the content of the cross-reference table, see “Using the compiler listing” on page 123. If both XREF and ATTRIBUTES are specified, the two listings are combined.

Default: NOXREF

Example:

```
noxref
```

Chapter 5. PL/I preprocessors

PL/I preprocessors

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

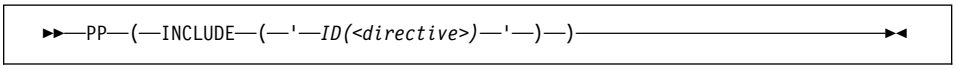
- The include preprocessor processes special include directives and incorporates external source files.
- The macro facility, based on %statements and macros, modifies your source program.
- The SQL preprocessor modifies your source program and translates EXEC SQL statements into PL/I statements.
- The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

Each preprocessor supports a number of options to allow you to tailor the processing to your needs. You can set the default options for each of the preprocessors by using the corresponding attributes in the configuration file.

Include preprocessor

The include preprocessor allows you to incorporate external source files into your programs by using include directives other than the PL/I directive %INCLUDE.

The following syntax diagram illustrates the options supported by the INCLUDE preprocessor:



ID Specifies the name of the include directive. Any line that starts with this directive as the first set of nonblank characters is treated as an include directive.

The specified directive must be followed by one or more blanks, an include member name, and finally an optional semicolon. Syntax for ddname(membername) is not supported.

In the following example, the first include directive is valid and the second one is not:

```
++include payroll
++include syslib(payroll)
```

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

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

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

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

Include preprocessor options environment variable

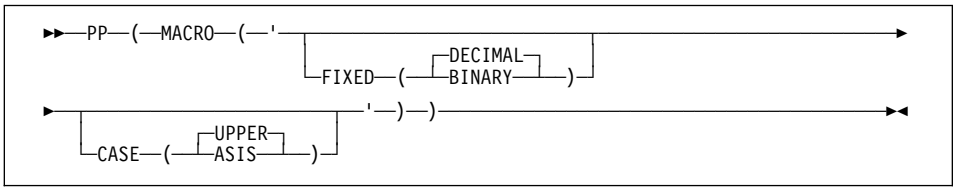
You can set the default options for the include preprocessor by using the IBM.PPINCLUDE environment variable. See “IBM.PPINCLUDE” on page 29.

Macro facility

Macros allow you to write commonly used PL/I code in a way that hides implementation details and the data that is manipulated, and only exposes the operations. In contrast with a generalized subroutine, macros allow generation of only the code that is needed for each individual use.

Macro facility options

You can specify `pp(macro)` without any options or include any of the following:



Abbreviations: DEC, BIN

CASE (ASIS or UPPER)
This option specifies whether the macro facility will be left "as is" or if it will be uppercased.

FIXED (DECIMAL or BINARY)
This option specifies the default base for FIXED variables as either DECIMAL or BINARY.

You can set the default options for the macro facility by using the `set IBM.PPMACRO` command. See “IBM.PPMACRO” on page 30.

Macro facility options environment variables

You can set the default options for the macro facility by using the IBM.PPMACRO environment variable. See “IBM.PPMACRO” on page 30.

SQL support

You can use dynamic and static EXEC SQL statements in PL/I applications. Before you can take advantage of EXEC SQL support, you must have installed IBM DB2 Universal Database (herinafter referred to as DB2) for Windows or IBM DB2 Universal Database for OS/2 depending on the operating system you are using. Workstation PL/I products support most of the function in DB2 and increased function will be added in each successive release. If you specify newer DB2 functions while using a downlevel DB2 product, warning messages are generated and those newer options are ignored.

Programming and compilation considerations

You need to consider specific options when using PL/I SQL support. The following table describes these considerations.

Table 3. Considerations for EXEC SQL support

If the target system is...	**Use this compile-time option...**
Windows using DB2 for Windows in native mode	DEFAULT (ASCII NATIVE IEEE)
OS/2 using DB2 for OS/2 in native PS/2 mode	DEFAULT with all of its defaults
CICS OS/2 using DB2 for OS/2 in native PS/2 mode	DEFAULT (ASCII NATIVE IEEE)
CICS VS/86 using DB2 for OS/2 in S/390 emulation mode	DEFAULT (EBCDIC NONNATIVE HEXADEC)
IMS using DB2 for OS/2 in S/390 emulation mode	DEFAULT (EBCDIC NONNATIVE HEXADEC)
ISPF dialog manager using DB2 for OS/2 in S/390 emulation mode	DEFAULT (EBCDIC NONNATIVE)

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

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

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

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

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

SQL support

SQL Users

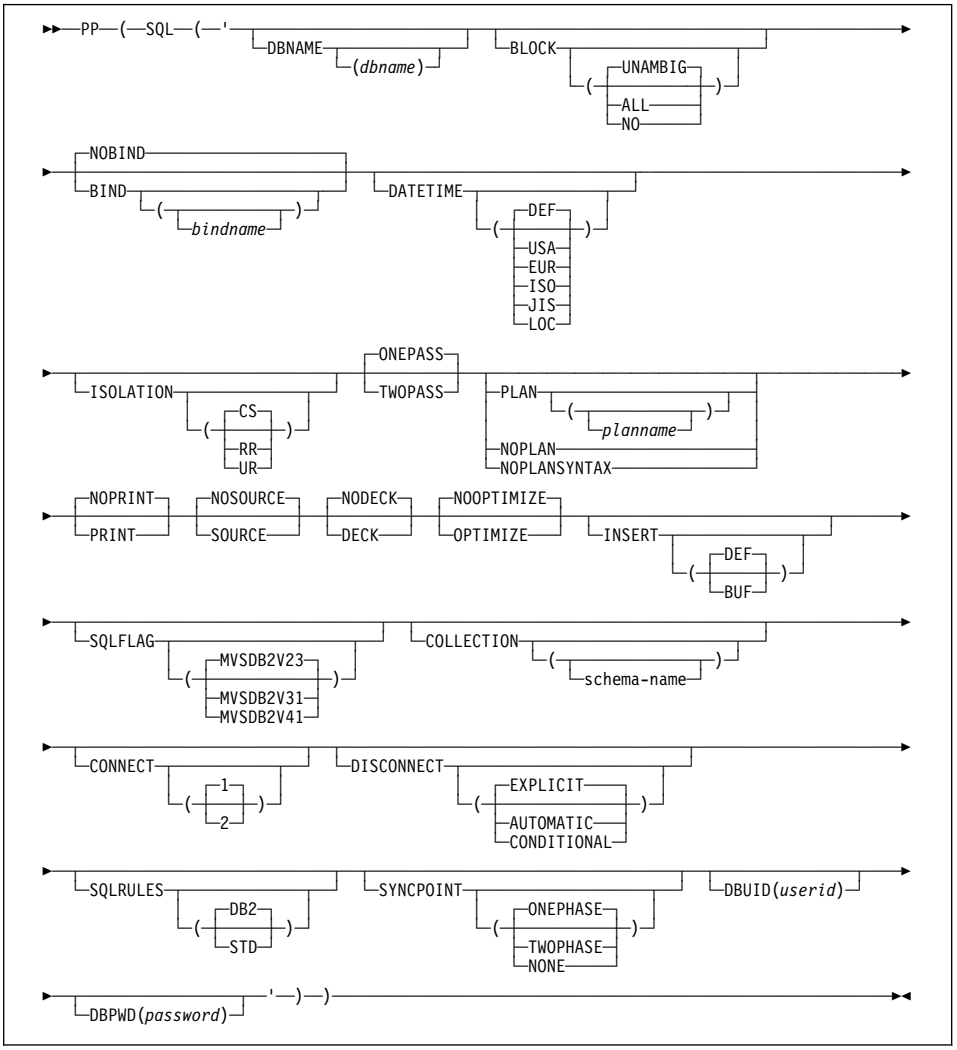
You must have DB2 for OS/2 or DB2 for Windows installed and started before you can compile a program containing EXEC SQL statements. To find out how to install DB2, refer to database installation guide for the platform you are using.

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

```
db2start
```

SQL preprocessor options

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



Abbreviations: DB, BLK, DT, ISOL, ON, TW, S, NS, D, ND, OPT, NOPT, INS, COL, CON, DISC, SQLR, SYNC

DBNAME

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

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

SQL support

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

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

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

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

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

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

DEF Use the date/time format associated with the country code of the database. This is also the default if DATETIME is not specified.
USA IBM standard for U.S. form.
Date format: mm/dd/yyyy
Time format: hh:mm xM (AM or PM)
EUR IBM standard for European form.
Date format: dd.mm.yyyy
Time format: hh.mm.ss
ISO International Standards Organization.
Date format: yyyy-mm-dd
Time format: hh.mm.ss
JIS Japanese Industrial Standards.
Date format: yyyy-mm-dd
Time format: hh:mm:ss
LOC Local form, not necessarily equal to DEF

ISOLATION
Specifies the isolation level at which your program runs.

CS Cursor stability
RR Repeatable read
UR Uncommitted read

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

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

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

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

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

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

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

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

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

BUF Use buffering when executing INSERTs with VALUES.

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

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

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

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

SQL support

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

COLLECTION
Specifies an eight character collection identifier for the package.

schema-name
Eight character identifier.

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

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

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

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

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

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

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

AUTOMATIC
Specifies that all database connections are disconnected at commit.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SQL preprocessor options environment variable

You can set the default options for the SQL Preprocessor by using the IBM.PPSQL environment variable. See “IBM.PPSQL” on page 30.

SQL support

SQL preprocessor BIND environment variables

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

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

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

Coding SQL statements in PL/I applications

You can code SQL statements in your PL/I applications using the language defined in *SQL Reference, Volume 1 and Volume 2* (SBOF-8923). Specific requirements for your SQL code are described in the sections that follow.

Defining the SQL communications area

A PL/I program that contains SQL statements must include an SQL communications area (SQLCA) As shown in Figure 1 on page 99 part of an SQLCA consists of an SQLCODE variable and an SQLSTATE variable.

- The SQLCODE value is set by the Database Manager after each SQL statement is executed. An application can check the SQLCODE value to determine whether the last SQL statement was successful.
- The SQLSTATE variable can be used as an alternative to the SQLCODE variable when analyzing the result of an SQL statement. Like the SQLCODE variable, the SQLSTATE variable is set by the Database Manager after each SQL statement is executed.

The SQLCA should be included by using the SQL INCLUDE statement:

```
exec sql include sqlca;
```

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

```
Dcl
  1 Sqlca,
    2 sqlcaid     char(8),           /* Eyecatcher = 'SQLCA   '*/
    2 sqlcabc     fixed binary(31),  /* SQLCA size in bytes = 136 */
    2 sqlcode     fixed binary(31),  /* SQL return code */
    2 sqlerrm     char(70) var,      /* Error message tokens */
    2 sqlerrp     char(8),           /* Diagnostic information */
    2 sqlerrd(6)  fixed binary(31),  /* Diagnostic information */
    2 sqlwarn,                       /* Warning flags */
      3 sqlwarn0  char(1),
      3 sqlwarn1  char(1),
      3 sqlwarn2  char(1),
      3 sqlwarn3  char(1),
      3 sqlwarn4  char(1),
      3 sqlwarn5  char(1),
      3 sqlwarn6  char(1),
      3 sqlwarn7  char(1),
    2 sqlext,
      3 sqlwarn8  char(1),
      3 sqlwarn9  char(1),
      3 sqlwarna  char(1),
      3 sqlstate  char(5);       /* State corresponding to SQLCODE */
```

Figure 1. The PL/I declaration of SQLCA

Defining SQL descriptor areas

The following statements require an SQLDA:

PREPARE *statement-name* INTO *descriptor-name* FROM *host-variable*
EXECUTE...USING DESCRIPTOR *descriptor-name*
FETCH...USING DESCRIPTOR *descriptor-name*
OPEN...USING DESCRIPTOR *descriptor-name*
DESCRIBE *statement-name* INTO *descriptor-name*

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

```
exec sql include sqlda;
```

The SQLDA must not be defined within an SQL declare section.

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

Figure 2. The PL/I declaration of an SQL descriptor area

Embedding SQL statements

The first statement of your PL/I program must be a PROCEDURE or a PACKAGE statement. You can add SQL statements to your program wherever executable statements can appear. Each SQL statement must begin with EXEC (or EXECUTE) SQL and end with a semicolon (;).

For example, an UPDATE statement might be coded as follows:

```
exec sql update Department
         export   Mgrno  = :Mgr_Num
         where    Deptno = :Int_Dept;
```

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

Continuation for SQL statements: The line continuation rules for SQL statements are the same as those for other PL/I statements.

Including code: SQL statements or PL/I host variable declaration statements can be included by placing the following SQL statement at the point in the source code where the statements are to be embedded:

```
exec sql include member;
```

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

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

Statement labels: With the exception of the END DECLARE SECTION statement, and the INCLUDE text-file-name statement, executable SQL statements, like PL/I statements, can have a label prefix.

WHENEVER statement: The target for the GOTO clause in an SQL WHENEVER statement must be a label in the PL/I source code and must be within the scope of any SQL statements affected by the WHENEVER statement.

Using host variables

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

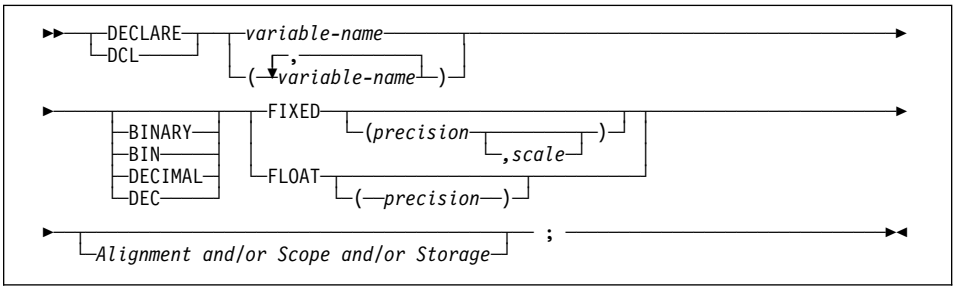
- All host variables within an SQL statement must be preceded by a colon (:).
- The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.
- An SQL statement that uses a host variable must be within the scope of the statement in which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

Declaring host variables: Host variable declarations can be made at the same place as regular PL/I variable declarations.

Only a subset of valid PL/I declarations are recognized as valid host variable declarations. The preprocessor does not use the data attribute defaults specified in the PL/I DEFAULT statement. If the declaration for a variable is not recognized, any statement that references the variable might result in the message **"**`The host variable token ID is not valid`**"**.

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

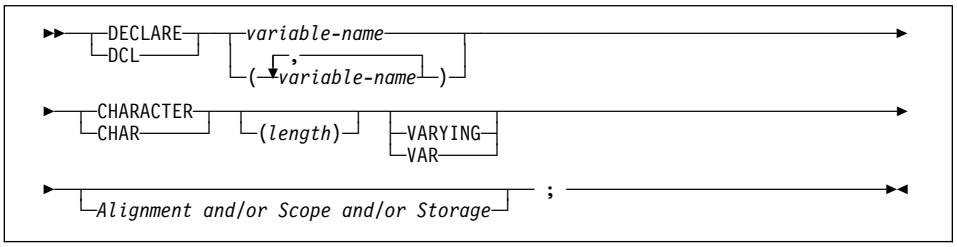
Numeric host variables: The following figure shows the syntax for valid numeric host variable declarations.



Notes

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

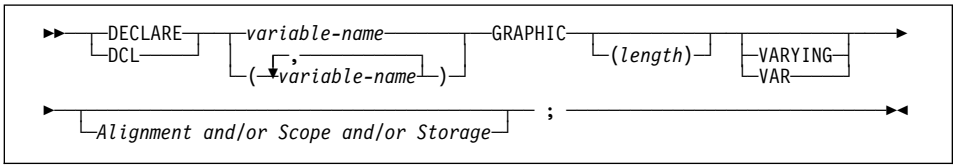
Character host variables: The following figure shows the syntax for valid character host variables.



Notes

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

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



Notes

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

Determining equivalent SQL and PL/I data types

The base SQLTYPE and SQLLEN of host variables are determined according to the following table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

Table 4. SQL data types generated from PL/I declarations

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

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

SQL support

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

Table 5. SQL data types mapped to PL/I declarations

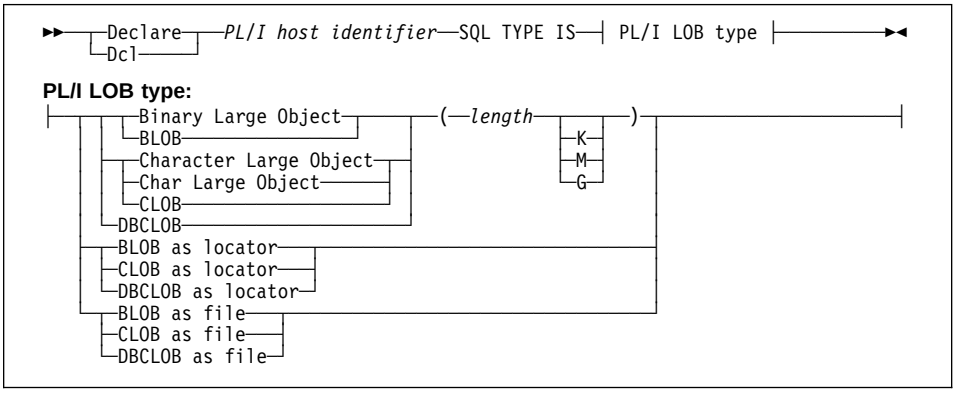
SQL Data Type	PL/I Equivalent	Notes
SMALLINT	BIN FIXED(15)	
INTEGER	BIN FIXED(31)	
DECIMAL(p,s)	DEC FIXED(p) or DEC FIXED(p,s)	p = precision and s = scale; 1 ≤ p ≤ 31 and 0 ≤ s ≤ p
FLOAT	BIN FLOAT(p) or DEC FLOAT(m)	22 ≤ p ≤ 53 7 ≤ m ≤ 16
CHAR(n)	CHAR(n)	1 ≤ n ≤ 254
VARCHAR(n)	CHAR(n) VAR	1 ≤ n ≤ 4000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

Large Object (LOB) support

Binary Large Objects (BLOBs), Character Large Objects (CLOBs), and Double Byte Character Large Objects (DBCLOBs), along with the concepts of LOB LOCATORS and LOB FILES are now recognized by the preprocessor. Refer to the DB2 manuals for more information on these subjects,

General information on LOBs

LOBS, CLOBS, and BLOBS can be as large as 2,147,483,640 bytes long (2 Gigabytes - 8 bytes for PL/I overhead). Double Byte CLOBS can be 1,073,741,820 characters long (1 Gigabyte - 4 characters for PL/I overhead). BLOBS, CLOBS, AND DBCLOBS can be declared in PL/I programs with the following syntax (*PL/I variables for Large Object columns, locators, and files*):



BLOB, CLOB, and DBCLOB data types

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

For example, consider the following declare:

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

The SQL preprocessor would transform the declare into this structure:

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

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

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

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

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

length

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

BLOB, CLOB, and DBCLOB LOCATOR data types

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

For example, consider the following declare:

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

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

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

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

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

BLOB, CLOB, and DBCLOB FILE data types

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

For example, consider this declare:

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

The SQL preprocessor transforms the declare as follows:

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

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

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

PL/I variable declarations for LOB Support

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

Example 1

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

After transform:

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

Example 2

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

After transform:

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

Example 3

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

After transform:

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

Example 4

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

After transform:

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

Dcl my_blob_file type blob_file;
```

Example 5

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

After transform:

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

Dcl my_dbclob_file type dbclob_file;
```

Sample programs for LOB support

Three sample programs are provided to show how LOB types can be used in PL/I programs:

SQLLOB1.PLI
Shows how to fetch a BLOB from the database into a file.

SQLLOB2A.PLI
Shows how to use LOCATOR variables to modify a LOB without any movement of bytes until the final assignment of the LOB expression.

SQL support

SQLLOB2B.PLI
Fetches the CLOB created in SQLLOB2A.PLI into a file for viewing.

User defined functions sample programs

You must install the following items to access the User Defined Function (UDF) sample programs:

- DB2 V2.1 or later
- Sample database

Several PL/I programs have been included to show how to code and use UDFs. Here is a short description of how to use them.

The file `UDFDLL.PLI` contains five sample UDFs. While these are simple in nature, they show basic concepts of UDFs.

MyAdd Adds two integers and returns the result in a third integer.

MyDiv Divides two integers and returns the result in a third integer.

MyUpper Changes all lowercase occurances of a,e,i,o,u to uppercase.

MyCount Simple implementation of counter function using a scratchpad.

ClobUpper Changes all lowercase occurences of a,e,i,o,u in a CLOB to uppercase then writes them out to a file.

Use the command file `bldudfdll` to compile and link it into the `udfdll` library.

After the `udfdll` library has been compiled and linked, copy it to the user defined function directory for your database instance. If you are using PL/I Set for AIX, for example, you would copy `udfdll` to `/u/inst1/sqllib/function` if that were the user defined function directory on your AIX machine for your database instance.

Before the functions can be used they must be defined to DB2. This is done using the CREATE FUNCTION command. The sample program, `addudf.pli`, has been provided to perform the CREATE FUNCTION calls for each UDF. CREATE FUNCTION calls would look something like the following:

```
CREATE FUNCTION MyAdd ( INT, INT ) RETURNS INT NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyAdd'

CREATE FUNCTION MyDiv ( INT, INT ) RETURNS INT NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyDiv'

CREATE FUNCTION MyUpper ( VARCHAR(61) ) RETURNS VARCHAR(61) NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyUpper'

CREATE FUNCTION MyCount ( ) RETURNS INT NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!MyCount'
  SCRATCHPAD

CREATE FUNCTION ClobUpper ( CLOB(5K) ) RETURNS CLOB(5K) NO SQL
  LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
  STYLE DB2SQL EXTERNAL NAME 'udfdll!ClobUpper'
```

These are just sample CREATE FUNCTION commands. Consult your DB2 manuals for more information or refinement.

Use the command file `bldaddudf` to compile and link the `addudf.pli` program. After it is compiled and linked, run it to define the user defined functions to your database.

Several sample PL/I programs are provided that call the user defined functions you have just created and added to the database:

UDFMYADD.PLI Fetches `ID` and `Dept` from the STAFF table then adds them together by calling `MyAdd` UDF. Use the command file `bldmyadd` to compile and link it.

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

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

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

UDFCLOB.PLI Fetches the resume for employee '000150' then calls `ClobUpper` to change the vowels to uppercase. Use the command file `bldclobu` to compile and link it. After this progam is run, look in the file `udfclob.txt` for the results.

Once these sample progams are compiled, linked, and the UDFs defined to DB2, the PL/I programs can be run from the command line.

These UDFs may also be called from the DB2 Command Line just line any other builtin DB2 function. For further information on how to customize and get the most out of your UDFs, please refer to your DB2 manuals.

Determining compatibility of SQL and PL/I data types

PL/I host variables in SQL statements must be type compatible with the columns which use them:

- Numeric data types are compatible with each other. A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a PL/I host variable of BIN FIXED(15), BIN FIXED(31), DECIMAL(*p*,*s*), BIN FLOAT(*n*) where *n* is from 22 to 53, or DEC FLOAT(*m*) where *m* is from 7 to 16.
- Character data types are compatible with each other. A CHAR or VARCHAR column is compatible with a fixed-length or varying-length PL/I character host variable.

  Graphic data types are compatible with each other. A GRAPHIC or VARGRAPHIC column is compatible with a fixed-length or varying-length PL/I graphic character host variable.
- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.

  When necessary, the Database Manager automatically converts a fixed-length character string to a varying-length string or a varying-length string to a fixed-length character string.

Using host structures

A PL/I host structure name can be a structure name with members that are not structures or unions. For example:

```
dcl 1 A,
      2 B,
        3 C1 char(...),
        3 C2 char(...);
```

In this example, B is the name of a host structure consisting of the scalars C1 and C2.

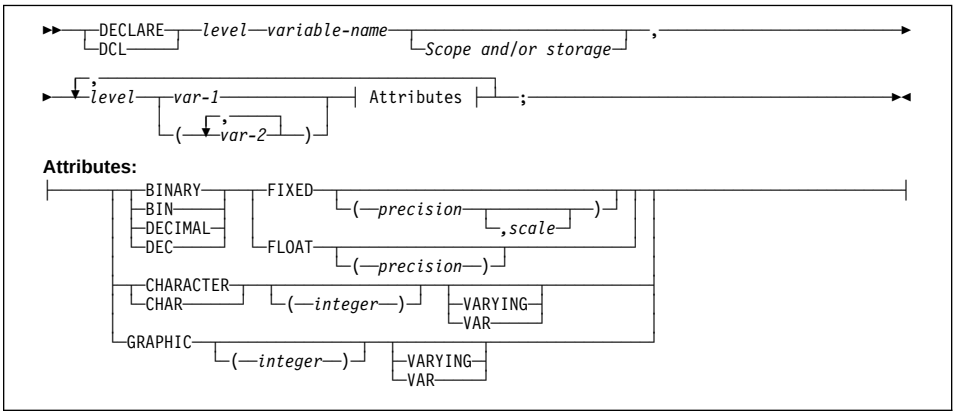
Host structures are limited to two levels. A host structure can be thought of as a named collection of host variables.

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

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

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

The following diagram shows the syntax for valid host structures.



Using indicator variables

An indicator variable is a two-byte integer (BIN FIXED(15)). On retrieval, an indicator variable is used to show whether its associated host variable has been assigned a null value. On assignment to a column, a negative indicator variable is used to indicate that a null value should be assigned.

Indicator variables are declared in the same way as host variables and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

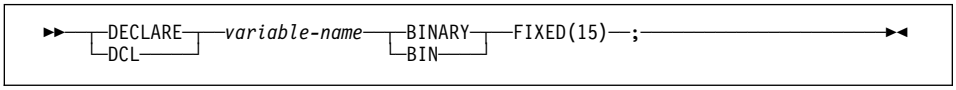
Given the statement:

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

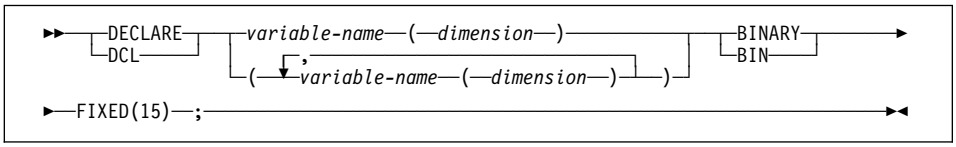
Variables can be declared as follows:

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

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



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



Host structure example

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

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


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


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

CONNECT TO statement

You can use a host variable to represent the database name you want your application to connect to, for example:

```
exec sql connect to :dbase;
```

If a host variable is specified:

- It must be a character or a character varying variable.
- It must be preceded by a colon and must not be followed by an indicator variable.
- The server-name that is contained within the host variable must be left-justified.

- If the length of the server name is less than the length of the fixed-length character host variable, it must be padded on the right with blanks.

```
dcl dbase char (10);
dbase = 'SAMPLE';            /* blanks are padded automatically */
exec sql connect to :dbase;
```

- If a varying character host variable is used, you may receive the following warning from the compiler. You can ignore this message.

```
IBM1214I W   xxx.x   A dummy argument is created for argument
                     number 6 in entry reference SQLESTRD_API
```

DECLARE TABLE statement

The preprocessor ignores all DECLARE TABLE statements.

DECLARE STATEMENT statement

The preprocessor ignores all DECLARE STATEMENT statements.

Logical NOT sign (¬)

The preprocessor performs the following translations within SQL statements:

¬= is translated to <>

¬< is translated to >=

¬> is translated to <=

Handling SQL error return codes

PL/I provides a sample program `DSNTIAR.PLI` that you can use to translate an SQLCODE into a multi-line message for display purposes. This PL/I program provides the same function as the DSNTIAR program on mainframe DB2*.

You must compile DSNTIAR with the same DEFAULT and SYSTEM compile-time options that are used to compile the programs that use DSNTIAR.

- If you are using DSNTIAR in native OS/2 or Windows PL/I programs, DSNTIAR must be compiled with following compile-time options:
  - DEFAULT(ASCII NATIVE LINKAGE(OPTLINK))
  - SYSTEM(OS/2) if you are using OS/2
  - SYSTEM(WINDOWS) if you using Windows
- If you are using DSNTIAR in host emulation PL/I programs, DSNTIAR must be compiled with DEFAULT(EBCDIC NONNATIVE LINKAGE(SYSTEM)) and SYSTEM(MVS) compile-time options.

The caller must declare the entry and conform to the interface as described in the mainframe DB2 publications. For your information, the declaration is of the following form:

```
dcl dsntiar entry options(asm inter retcode);
```

Three arguments are always passed:

arg 1
This input argument must be the SQLCA.

arg 2
This input/output argument is a structure of the form:

```
dcl 1 Message,
      2 Buffer_length fixed bin(15) init(n), /* input  */
      2 User_buffer char(n);         /* output */
```

You must fill in the appropriate value for *n*.

arg 3
This input argument is a FIXED BIN(31) value that specifies logical record length.

Use of varying strings under DFT(EBCDIC NONNATIVE)

If you specify the compile-time option DFT(EBCDIC NONNATIVE) and you use a varying string host variable as input to the database, you must initialize the host variable or you might get a protection exception during the execution of your program.

If you use an uninitialized varying string on mainframe DB2, your program would be in error and might also get a protection exception.

Using the DEFAULT(EBCDIC) compile-time option

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

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

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

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

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

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

Avoiding automatic conversion using DCLGEN: Another way to avoid the conversion caused by using DEFAULT(EBCDIC) is to use the DCLGEN utility that is provided with VisualAge PL/I to create the declares for database tables.

DCLGEN automatically generates the comment directive required in the output when it recognizes that a column is defined with the FOR BIT DATA attribute.

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

SQL compatibility and migration considerations

The workstation compilers tolerate the following statement:

```
' EXEC SQL CONNECT :userid IDENTIFIED BY :passwd'
```

The preceding statement is translated by the PL/I SQL preprocessor and sent to the database precompiler services as:

```
' EXEC SQL CONNECT'
```

The allows VM SQL/DS users to compile their programs without making significant changes.

CICS support

If you do not specify the PP(CICS) option, EXEC CICS statements are parsed and variable references in them are validated. If they are correct, no messages are issued as long as the NOCOMPILE option is in effect. Without invoking the CICS preprocessor, real code cannot be generated.

You can use EXEC CICS statements in PL/I applications that run as transactions under CICS.

You can develop these applications under CICS on OS/2 or Windows for eventual execution under CICS that particular development platform or under CICS/ESA, CICS/MVS, or CICS/VSE systems on S/390.

OS/2 Make sure that the CICS installation adds all the `\OPT\...` settings to your system environment variables for Windows support and updates your CONFIG.SYS to include `\cicsnnn\bin` in the LIBPATH for your OS/2 support. Where `cicsnnn` is the root level for CICS OS2. It is not necessary that the CICS system be operational when you are compiling your programs.

Programming and compilation considerations

When you are developing programs for execution under CICS:

- You must use the SYSTEM(CICS) compile-time option.
- You must use the PP(CICS(*options*) MACRO) compile-time option. The MACRO option must follow the CICS option of PP.

If your CICS programs include files or use macros that contain EXEC CICS statements, you must also use either the MACRO compile-time option or the MACRO option of PP before the CICS option of the PP option as shown in the following example:

```
pp (macro(...)  cics(...) macro(...) )
```

Make sure that INC is specified as an extension on the INCLUDE(EXT) compile-time option, see “INCLUDE” on page 53.

The IBM.SYSLIB or INCLUDE environment variable must specify the CICS include file directories, for example:

```
set include=d:\cicsnnn\plihdr;
```

The PL/I declarations generated by the CICSMAP, the Basic Mapping Support (BMS) utility, are placed in the first directory specified in the INCLUDE environment variable. For more information, see “Setting compile-time environment variables” on page 28.

Output produced in one of the following ways is written to the CPLI transient data queue (TDQ):

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

Output produced by PLIDUMP is always written to the CPLD transient data queue.

The full workstation CICS API is supported for PL/I programs. Support is also provided for PL/I progams to use:

- External Presentation Interface (EPI)
- External Call Interface (ECI)
- External Transaction Initiation (ETI)

Other PL/I considerations that apply on S/390 CICS apply to CICS on the workstation also. The program behaves as though the STAE option is always in effect. The NOSTAE option is not supported.

If you are developing applications for eventual execution on S/390 CICS subsystems, you can check your PL/I programs for reentrancy violations with the DEFAULT(NONASSIGNABLE) compile-time option.

For compatibility with CICS/ESA, CICS/MVS, and CICS/VSE, make sure that the EXEC CICS commands are in upper case.

You can use PL/I FETCH and RELEASE under CICS.

A CICS program must not have more than one procedure that has OPTIONS(MAIN).

The EXEC CICS ADDRESS and other similar commands that return a pointer to a CICS control block (such as the TWA COMMAREA, and ACEE) might return a SYSNULL() pointer if the control block does not exist. (For example, '00000000'x not 'FF000000'x) Your programs must use the SYSNULL built-in function to test such pointers.

Each PL/I compilation unit processed by the CICS preprocessor generates the following:

```
dcl IBMMCICS_ID char(n) static init('cics-id-and-version');
```

The name, version, and release level of the CICS system for which your program was compiled are indicated.

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

Table 6 (Page 1 of 2). Considerations for EXEC CICS support

If you are using ...	Use compile-time option(s)...
CICS for OS/2 or Windows	PP(CICS MACRO)
CICS Files containing native PS/2 data	DEFAULT (ASCII NATIVE IEEE) as appropriate
DB2/2 in native PS/2 mode	DEFAULT (ASCII NATIVE IEEE) as appropriate

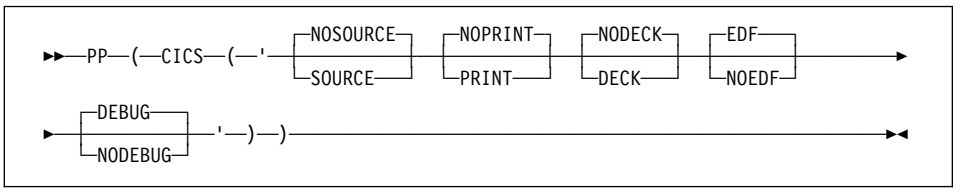
Table 6 (Page 2 of 2). Considerations for EXEC CICS support

If you are using ...	Use compile-time option(s)...
CICS Files containing host S/390 data	DEFAULT (EBCDIC NONNATIVE HEXADEC) as appropriate
DB2/2 in host S/390 mode	DEFAULT (EBCDIC NONNATIVE HEXADEC) as appropriate

You must have CICS installed before you can compile a program containing EXEC CICS statements. To find out how to install CICS on your workstation, refer to the installation instructions for that product.

CICS preprocessor options

The following syntax diagram show options supported by the CICS preprocessor.



Abbreviations: S, NS, D, ND

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

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

DECK or NODECK
Specifies that the CICS preprocessor output source is written to a file with the extension .DEK. The file is in the current directory.

EDF or NOEDF
Specifies whether or not the CICS Execution Diagnostic Facility (EDF) is to be enabled for the PL/I program. There is no performance advantage in specifying NOEDF, but the option can be useful in preventing CICS commands from appearing on EDF displays in well tested programs.

DEBUG or NODEBUG
Specifies whether or not the CICS preprocessor is to pass source program line numbers to CICS for use by the CICS Execution Diagnostic Facility (EDF).

CICS preprocessor options environment variables

You can set the default options for the CICS preprocessor by using the IBM.PPCICS environment variable. See “IBM.PPCICS” on page 30.

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*, SC33-1568. Specific requirements for your CICS code are described in the sections that follow.

Embedding CICS statements

The first statement of your PL/I program must be a PROCEDURE statement. You can add CICS statements to your program wherever executable statements can appear. Each CICS statement must begin with EXEC (or EXECUTE) CICS and end with a semicolon (;).

For example, the GETMAIN statement might be coded as follows:

```
exec cics getmain set(blk_ptr) length(stg(blk));
```

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

Continuation for CICS statements: Line continuation rules for CICS statements are the same as those for other PL/I statements.

Including code: If included code contains EXEC CICS statements or your program uses PL/I macros that generate EXEC CICS statements, you must use one of the following:

- The MACRO compile-time option
- The MACRO option of the PP option (before the CICS option of the PP option)

Margins: CICS statements must be coded within the columns specified in the MARGINS compile-time option.

Statement labels: EXEC CICS statements, like PL/I statements, can have a label prefix.

Writing CICS transactions in PL/I

This section describes the rules and guidelines that apply to PL/I support of CICS on the workstation.

You can use PL/I with CICS facilities to write application programs (transactions) for CICS subsystems. If you do this, CICS provides facilities to the PL/I program that would normally be provided directly by the operating system. These facilities include most data management facilities and all job and task management facilities.

You should observe the following rules to ensure compatibility with S/390 PL/I CICS support.

- Do not use macro level support, only command level support is provided.
- Do not use any PL/I input or output except:

  Stream output for SYSPRINT
  PLIDUMP

  Since these are intended for debugging purposes only, you should not include them in production programs for performance reasons.
- Do not use the following statements:

  DELAY
  WAIT
- You should not communicate with FORTRAN, COBOL, or C, using PL/I interlanguage facilities. However, CICS programs written in different languages can communicate with each other using EXEC CICS LINK or XCTL commands.

  Subroutines written in a language other than PL/I can be called using PL/I interlanguage facilities providing those subroutines do not contain any EXEC CICS code. If you want to communicate with a non-PL/I program that contains EXEC CICS code, you must use EXEC CICS LINK or EXEC CICS XCTL as stated.

  COBOL and C are supported under CICS by the following IBM PL/I products:

  IBM PL/I MVS and VM
  IBM PL/I Set for AIX
  IBM VisualAge PL/I
- Do not use the PLISRTx built-in subroutines.
- Do not make calls to IMS using the PLITDLI, ASMTDLI, or EXEC DLI.

CICS abends used for PL/I programs

APLS

This abend is issued on termination, if termination is caused by the ERROR condition, and the ERROR condition was not caused by an abend (other than an ASRA abend).

This is the abend code issued by PL/I when either:

1. A transaction terminates in error due to a PL/I software interrupt (CONVERSION, for example), and there is no ERROR ON-unit
2. The program takes normal return from the ERROR ON-unit.

Because the program failed, the failure must be reflected to CICS on your workstation as an abend so that DTB, and so on, can occur if necessary.

APLT

An error was detected in the user exit.

CICS run-time user exit

It is strongly recommended that you review and modify (if necessary) the IBM-supplied CICS user exit, CEEFXITA. See “Using the CICS run-time user exit” on page 383.

Chapter 6. Compilation output

The results of compilation depend on how error-free your source program is and on the compile-time options you specify. Results can include diagnostic messages, a return code, and other output saved to disk (for example, an object module and a listing). The following section describes a sample compiler listing. “Compiler output files” on page 130 describes other kinds of output files you can request from the compiler.

Using the compiler listing

During compilation, the compiler generates listings that contain information about the source program, the compilation, and the object module. The TERMINAL option sends diagnostics and statistics to your terminal. The IBM.PRINT environment variable specifies the output directory for printable listing files (see “IBM.PRINT” on page 31 for more information on the IBM.PRINT environment variable). The following description of the listing refers to its appearance on a printed page.

This listing for CHIMES program highlights some of the more useful parts of the compiler listing. Figure 3 is similar to the compiler listing for that program.

```
5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 1


                  Options Specified  1

 Install:
 Command: number options a(s) x nest gonumber lc(55)

 Line.File Process Statements

    1.1    *PROCESS MACRO S A(F) X AG;
    2.1    *PROCESS LANGLVL(SAA2);
    3.1    *PROCESS NOT('¬') OR('|');
```

Figure 3 (Part 1 of 5). CHIMES program compiler listing

Using the compiler listing

```
5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 2

                  Options Used  2

 +    AGGREGATE
 +    ATTRIBUTES(FULL)
      CHECK( NOSTORAGE )
    NOCOMPILE(S)
      CURRENCY('$')
      DEFAULT(IBM ASSIGNABLE NOINITFILL
              NONCONNECTED DESCRIPTOR NODUMMYDESC
              BYADDR RETURNS(BYVALUE) LINKAGE(OPTLINK)
              NOINLINE ORDER NONRECURSIVE
              NULLSYS EVENDEC NORETCODE
              ASCII IEEE NATIVE NATIVEADDR)
    NODLLINIT
    NOEXIT
      FLAG(W 250)
 +    GONUMBER
    NOGRAPHIC
      IMPRECISE
      INCAFTER(PROCESS(""))
    NOINSOURCE
 +    LANGLVL(SAA2 NOEXT)
      LIBS
      LIMITS(EXTNAME(7) FIXEDDEC(15))
 +    LINECOUNT(55)
    NOLIST
 +    MACRO
      MARGINI(' ')
      MARGINS(2 72)
    NOMDECK
      NAMES('@#$' '@#$')
      NATLANG(ENU)
 +    NEST
 +    NOT('¬')
      OBJECT
      OPTIMIZE(0)
 +    OPTIONS
      OR('|')
 +    PP(MACRO)
    NOPPTRACE
      PREFIX(CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW
             NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE
             UNDERFLOW ZERODIVIDE)
      PROBE
    NOPROCEED(S)
    NOPROFILE
      RESPECT()
      RULES(IBM NOLAXDCL NOLAXIF LAXQUAL NOLAXCOMMENT)
    NOSEMANTIC(S)
    NOSNAP
 +    SOURCE
    NOSYNTAX(S)
      SYSPARM('')
      SYSTEM(OS2 S386)
      TERMINAL
    NOTEST
    NOTILED
```

Figure 3 (Part 2 of 5). CHIMES program compiler listing

```
5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 3

      TRACK()
      XINFO(NODEF NOSYNTAX)
 +    XREF
      WINDOW(1950)

5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 4
 Line.File LV NT


    5.1            chimes: proc options(main);      /* Play a tune using DOSBEEP tones */

    7.1     1         dcl ( rest  value(   0 ),            /* Declare Named Constants  */
                            g4    value( 392 ),            /* for note and rest tone   */
                            c5    value( 523 ),            /* values and timings.      */
                            d5    value( 587 ),
                            e5    value( 657 ),
                            whole value( 800 ) ) fixed bin(31);

   14.1     1         dcl notes(19,2) static nonasgn fixed bin(31)
    3               init( e5, (whole/2),              /* Declare tone and timing */
                             c5, (whole/2),              /* for each note of tune.  */
                             d5, (whole/2),
                             g4, (whole),                /* Initial values may be   */
                           rest, (whole/2),              /* restricted expressions  */
                             g4, (whole/2),              /* using Named Constants   */
                             d5, (whole/2),              /* previously defined in   */
                             e5, (whole/2),              /* this program.           */
                             c5, (whole),
                           rest, (whole/2),
                             e5, (whole/2),
                             c5, (whole/2),
                             d5, (whole/2),
                             g4, (whole),
                           rest, (whole/2),
                             g4, (whole/2),
                             d5, (whole/2),
                             e5, (whole/2),
                             c5, (whole) );

   35.1     1         dcl i fixed bin(31);

                      /* Declare external APIs called by chimes. */

   51.1     1         dcl beep     entry( fixed bin(31), fixed bin(31) ) /* tone, time */
                                   returns( fixed bin(31) optional )  /* Ignore return */
                                   ext( 'DOSBEEP' )       /* External name of function */
                                   options( byvalue
                                            linkage(system));


   58.1     1         dcl sleep    entry( fixed bin(31) )        /* Time duration only */
                                   returns( fixed bin(31) optional )
                                   ext( 'DOSSLEEP' )
                                   options( byvalue
                                            linkage(system) );

                      /* Play all of the notes and rests of the tune using a do loop.  */

   67.1     1         do i = lbound(notes,1) to hbound(notes,1);
   68.1     1  1        if notes(i,1) ¬= 0        /* Note the use of ¬ for logical NOT */
```

Figure 3 (Part 3 of 5). CHIMES program compiler listing

Using the compiler listing

```
5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 5
 Line.File LV NT

   69.1     1  1        then call beep( notes(i,1), notes(i,2) );
   70.1     1  1        else call sleep( notes(i,2) );
   71.1     1  1      end;

   73.1     1      end;


5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 6


                 Attribute/Xref Table  4


    Line.File Identifier                    Attributes

      51.1    BEEP                          CONSTANT EXTERNAL('DOSBEEP')
                                            ENTRY( BYVALUE FIXED BIN(31,0),
                                                   BYVALUE FIXED BIN(31,0) )
                                            RETURNS( BYVALUE OPTIONAL FIXED
                                                     BIN(31,0) )
                                            Refs: 69.1
       9.1    C5                            CONSTANT FIXED BIN(31,0)
                                            Refs: 14.1 14.1 14.1 14.1
       5.1    CHIMES                        CONSTANT EXTERNAL
                                            ENTRY()
      10.1    D5                            CONSTANT FIXED BIN(31,0)
                                            Refs: 14.1 14.1 14.1 14.1
      11.1    E5                            CONSTANT FIXED BIN(31,0)
                                            Refs: 14.1 14.1 14.1 14.1
       8.1    G4                            CONSTANT FIXED BIN(31,0)
                                            Refs: 14.1 14.1 14.1 14.1
     +++++++  HBOUND                        BUILTIN
                                            Refs: 67.1
      35.1    I                             AUTOMATIC FIXED BIN(31,0)
                                            Refs: 68.1 69.1 69.1 70.1
                                            Sets: 67.1
     +++++++  LBOUND                        BUILTIN
                                            Refs: 67.1
      14.1    NOTES                         STATIC NONASSIGNABLE
                                            DIM(1:19,1:2) FIXED BIN(31,0)
                                            INITIAL
                                            Refs: 67.1 67.1 68.1 69.1 69.1
                                                  70.1
       7.1    REST                          CONSTANT FIXED BIN(31,0)
                                            Refs: 14.1 14.1 14.1
      58.1    SLEEP                         CONSTANT EXTERNAL('DOSSLEEP')
                                            ENTRY( BYVALUE FIXED BIN(31,0) )
                                            RETURNS( BYVALUE OPTIONAL FIXED
                                                     BIN(31,0) )
                                            Refs: 70.1
      12.1    WHOLE                         CONSTANT FIXED BIN(31,0)
                                            Refs: 14.1 14.1 14.1 14.1 14.1
                                                  14.1 14.1 14.1 14.1 14.1
                                                  14.1 14.1 14.1 14.1 14.1
                                                  14.1 14.1 14.1 14.1
```

Figure 3 (Part 4 of 5). CHIMES program compiler listing

```
5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 7


                  Aggregate Length Table  5

                                  Total         Base
    Line.File         Offset       Size         Size  Identifier


      14.1              0          152            4 NOTES

5639-D65 VisualAge PL/I           V2.R1.00   1998.04.09  15:02:14   Page 8


File Reference Table

  File    Included From  Name
             6
    1                   d:\ibmpli\samples\chimes.pli
 3

Component    Return Code    Messages (Total/Suppressed)    Time  7

MACRO            0               0  /  0                  2 secs
Compiler         0               0  /  0                 12 secs

End of compilation of CHIMES
```

Figure 3 (Part 5 of 5). CHIMES program compiler listing

1 Options specified
This section of the compiler listing shows any compile-time options you specified. Options shown under `Install:` are specified in your CONFIG.SYS or AUTOEXEC.BAT file using the IBM.OPTIONS environment variable. Options shown under `Command:` indicate that these options were specified on the command line when you invoked the compiler (there are no command options in this example). Options specified with the *PROCESS or %PROCESS statement are shown below the command options.

2 Options used
The compiler listing includes a list of all compile-time options used, including the default options. If an option is marked with a plus sign (+), the default has been changed. If any compile-time options contradict each other, the compiler uses the one with the highest priority. The following list shows which options the compiler uses, beginning with the highest priority:

- Options specified with the *PROCESS or %PROCESS statement.
- Options specified when you invoked the compiler with the PLI command.
- Install options installed either at installation time or by the IBM.OPTIONS environment variable (see “IBM.OPTIONS” on page 29 for more information on the IBM.OPTIONS environment variable).

3 Using the NUMBER option
The statement numbers shown are generated by the NUMBER option. In this case, the statement begins on the 14th line in file 1. The File Reference Table at

the bottom of the listing also shows that file 1 refers to D:\ibmpli\samples\chimes.pli.

By generating these statement numbers during compilation, you can locate lines that need editing (indicated in messages, for example) without having to refer to the listing.

4 Attribute and cross-reference table

If you specify the ATTRIBUTES option, the compiler provides an attribute table containing a list of the identifiers in the source program together with their declared and default attributes in the compiler listing. The FULL attribute lists all identifiers and attributes. If you specify the SHORT suboption for ATTRIBUTES, unreferenced identifiers are not listed.

If you specify the XREF option, the compiler prints a cross-reference table containing a list of the identifiers in the source program together with the `Line.File` number (the statement number inside the file and the file number, respectively) in which they appear in the compiler listing.

An identifier appears in the `Sets:` part of the cross-reference table if it is:

- The target of an assignment statement.
- Used as a loop control variable in DO loops.
- Used in the SET option of an ALLOCATE or LOCATE statement.
- Used in the REPLY option of a DISPLAY statement.

If there are unreferenced identifiers, they are displayed in a separate table (not shown in this example).

If you specify ATTRIBUTES and XREF (as in this example), the two tables are combined.

Explicitly-declared variables are listed with the number of the DECLARE statement in which they appear. Implicitly-declared variables are indicated by asterisks and contextually declared variables (HBOUND and LBOUND in this example) are indicated by plus (+) signs. (Undeclared variables are also listed in a diagnostic message.)

The attributes INTERNAL and REAL are never included; they can be assumed unless the respective conflicting attributes, EXTERNAL and COMPLEX, are listed.

For a file identifier, the attribute FILE always appears, and the attribute EXTERNAL appears if it applies; otherwise, only explicitly declared attributes are listed.

For an array, the dimension attribute is printed first. If the bound of an array is a restricted expression, the value of that expression is shown for the bound; otherwise an asterisk is shown.

If the length of a bit string or character string is a restricted expression, that value is shown, otherwise an asterisk is shown.

5 Aggregate length table

If you specified the AGGREGATE option, the compiler provides an aggregate length table in the compiler listing. The table shows how each aggregate in the

program is mapped. Table 7 on page 129 shows the headings for the aggregate length table columns and the description of each.

Table 7. Aggregate length table headings and description

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

6 File reference table

The **Included From** column of the File reference table indicates where the corresponding file from the **Name** column was included. The first entry in this column is blank because the first file listed is the source file. Entries in the **Included From** column show the line number of the include statement followed by a period and the file number of the source file containing the include.

7 Component, return code, diagnostic messages, time

The last part of the compiler listing consists of the following headings:

Component

Shows you which component or processor is providing the information. Either the macro facility, if invoked, or the compiler itself can provide you with informational messages.

Return code

Shows you the highest return code generated by the component, issued upon completion of compilation. Possible return codes are:

0 (Informational)

No warning messages detected (as in this example). The compiled program should run correctly. The compiler might inform you of a possible inefficiency in your code or some other condition of interest.

4 (Warning)

Indicates that the compiler found minor errors, but the compiler could correct them. The compiled program should run correctly, but might produce different results than expected or be significantly inefficient.

8 (Error)

Indicates that the compiler found significant errors, but the compiler could correct them. The compiled program should run correctly, but might produce different results than expected.

Compiler output files

12 (Severe error)
Indicates that the compiler found errors that it could not correct. If the program was compiled and an object module produced, it should not be used.

16 (Unrecoverable error)
Indicates an error-forced termination of the compilation. An object module was not successfully created.

Note: When coding CMD files for PL/I, you can use the return code to decide whether or not post-compilation procedures are performed.

Messages
Indicates:

- The number of messages issued, if any
- The number of messages suppressed, if any, because they were equal to or below the severity level set by the FLAG compile-time option.

Messages for the compiler, macro facility, SQL preprocessor, and run-time environment are listed and explained in *VisualAge PL/I Messages and Codes*.

Only messages of the severity above that specified by the FLAG option are issued. The messages, statements, and return code appear on your screen unless you specify the NOTERMINAL compile-time option.

Time
Shows you the total time the component took to process your program.

Compiler output files

If you compile a program using default options, an object module is created in the current directory. By altering compile-time options, you can request other output to be created in addition to the object module. Table 8 on page 131 lists other possible compilation outputs which are also located in the current directory by default.

Compiler output files

All compiler output files use the same file name as the main program file. The file extensions are specified in the following table.

Table 8. Possible compilation disk outputs

Output	File extension	How requested (compile-time option)	How relocated (environment variable)
Preprocessed source text	DEK	DECK option of appropriate preprocessor	IBM.DECK
Object module	OBJ	OBJECT	IBM.OBJECT
Object listing	ASM	LIST	IBM.PRINT
Template .DEF file	DEF	XINFO(DEF)	IBM.OBJECT
Message listing	XML	XINFO(XML)	IBM.OBJECT

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

Chapter 7. Linking your program

The following sections describe how to link object files produced by the compiler into either an executable program file (.EXE) or dynamic link library (.DLL).

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

Every .DLL that you build must have at least one module compiled with the DLLINIT compile-time option (see “DLLINIT” on page 50).

Starting the linker

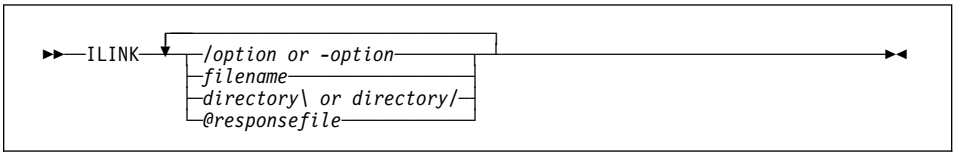
Once the compiler has created object modules out of your source files, use the linker to link them together with the PL/I runtime libraries to create an EXE or DLL file.

Statically linking

To statically link the library into your `.EXE`, specify the LIBS(SINGLE STATIC) or LIBS(MULTI STATIC) compile-time option (see “LIBS” on page 56). You must also link with the /NOE linker option.

Linking from the command line

Specify the ILINK command followed by any sequence of options, file names, or directories, separated by space or tab characters.



options
One or more ILINK options.

ILINK options start with a / or - character.

filename
The names of one or more of the following kinds of files:

- Object files—have an .OBJ filename extension
- Library files—have an .LIB filename extension
- Definition files—have a .DEF filename extension
- WIN Export files—have an .EXP filename extension
- WIN Resource files—have an .RES filename extension

You must specify at least one object file to use ILINK correctly.

directories
One or more directory locations which end with a / or \ character.

Starting the linker

responsefile
The name of a response file. The file name should immediately follow the `@` character.

Linking considerations

You can specify the name of the output file with the `/OUT` option. You can specify the name of a map file with the `/MAP` option.

In addition to the libraries you specify, by default the linker searches the PL/I runtime libraries defined in your object files at compile time (see “LIBS” on page 56).

The directories you specify become part of the linker's search path, before any directories set in the LIB environment variable. See “Search rules” on page 137 and “Specifying directories” on page 137 for more information.

You can use wildcard characters to specify multiple object files. For example, use *.OBJ to specify all the object files in a directory.

Filename extensions are not assumed

The linker does not assume extensions for files. If you specify a filename with no extension, then the linker looks for the file with that name and no extension. If the linker cannot find a file, it stops linking.

Examples

The following command links the object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. The linker searches for unresolved external references in the library file XLIB.LIB and in the default libraries. Since there is no name provided for the executable file, it is named FUN.EXE, taking the filename of the first object file and the default extension .EXE. The linker also produces a map file, FUNLIST.MAP.

```
ilink /MAP:funlist fun.obj text.obj table.obj care.obj xlib.lib
```

The following command links the files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ into an executable file named MAIN.EXE, and produces a map file named MAIN.MAP.

```
ilink /MAP main.obj getdata.obj printit.obj
```

OS/2 The following command links GETDATA.OBJ and PRINTIT.OBJ into a DLL named GETDATA.DLL.

```
ilink getdata.obj printit.obj /OUT:getdata.dll /DLL
```

WIN In Windows, the same command changes slightly by adding an export file, GETDATA.EXP, which specifies the functions that are exported from GETDATA.DLL.

```
ilink getdata.obj printit.obj /OUT:getdata.dll /DLL getdata.exp
```

Linking from a make file

Use a make file to organize the sequence of actions (such as compiling and linking) required to build your project. You can then invoke all the actions in one step. The NMAKE utility saves you time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files.

Linker input and output

The following figure contains a basic make file example.

```
#-------------------------------------------------------
#
#  fun.mak - sample makefile
#
#    Usage:  nmake fun.mak
#
#    The following commands are done only when needed:
#
#       - Compiles fun, text, table, care,
#                   xlib1, and xlib2
#       - Adds xlib1 and xlib2 to library xlib
#       - Links fun, text, table, care, and xlib
#          to build fun.exe
#
#    Each block is as follows:
#      <target>: <list of dependencies for target>
#           <action(s) required to build target>
#
#-------------------------------------------------------

OBJS = fun.obj text.obj table.obj care.obj
LIBS = xlib.lib

fun.exe: $(OBJS) $(LIBS)
  ilink /MAP:funlist $(OBJS) $(LIBS)

xlib.lib: xlib1.obj xlib2.obj
  ilib /OUT:xlib.lib xlib1.obj xlib2.obj
#    for OS/2, use:  ilib xlib -+xlib1 -+xlib2;

fun.obj:  fun.pli
  pli fun.pli

text.obj:  text.pli
  pli text.pli

table.obj:  table.pli
  pli table.pli

care.obj:  care.pli
  pli care.pli

xlib1.obj:  xlib1.pli
  pli xlib1.pli

xlib2.obj:  xlib2.pli
  pli xlib2.pli
```

Figure 4. Make file example

OS/2 You can write the makefile yourself, or you can use WorkFrame to manage the makefile. When you build through WorkFrame, a makefile is created and maintained automatically.

Input and output

The linker is designed to link object files with other library files you specify to produce either an executable program file (.EXE) or a dynamic link library (.DLL).

Linker input and output

The linker optionally produces a map file, which provides information about the contents of the executable output.

Input	Output
options	*executable file* (.EXE or .DLL)
object files (*.OBJ)	*map file* (.MAP)
library files (*.LIB)	*return code*
module definition file (.DEF)	
(Windows)*export files* (*.EXP)	
(Windows)*resource files* (*.RES)	

Search rules

When searching for an object (.OBJ), library (.LIB), or module definition (.DEF) file, the linker looks in the following locations in this order:

1. The directory you specified for the file or the current directory if you did not give a path. Default libraries do not include path specifications.

   Note: If you specify a path with the file, the linker searches only that path.

2. Any directories entered by themselves on the command line (they must end with a slash (/) or backslash (\) character). See the section on “Specifying directories” for more information.
3. Any directories listed in the LIB environment variable.

If the linker cannot locate a file, it generates an error message and stops linking.

Example

A response file could contain the following information:

```
FUN.OBJ TEXT.OBJ TABLE.OBJ CARE.OBJ
NEWLIBV3.LIB
C:\TESTLIB\
```

The linker links four object files to create an executable file named FUN.EXE. The linker searches NEWLIBV3.LIB before searching the default libraries to resolve references.

To locate NEWLIBV3.LIB and the default libraries, the linker searches the following locations in this order:

1. The current directory (because NEWLIBV3.LIB was entered without a path)
2. The C:\TESTLIB\ directory
3. The directories listed in the LIB environment variable

Specifying directories

To have the linker search additional directories for input files, specify a drive or directory by itself on the command line. Specify the drive or directory with a slash (/) or backslash (\) character at the end so the linker will recognize it as a path.

Specifying object files

The linker searches the paths you specify before it searches the paths in the LIB environment variable. See the section on “Search rules” for more information.

Filename defaults

If you do not enter a file name, the linker assumes the following defaults:

Table 9. Linker filename defaults

File	Default Filename
Object files	None. You must enter at least one object file name.
Output file	The base name of the first object file.
Map file	The base name of the output file.
Library files	The default libraries defined in the object files. Use the LIBS compile-time option to define the default libraries. Any additional libraries you specify are searched before the default libraries.
Module definition file	None. The linker assumes you accept the default for all module statements.

Specifying object files

When you invoke the linker from the command line, the linker assumes that any input it cannot recognize as other files, options, or directories must be an object file. Use a space or tab character to separate files. See “Linking from the command line” on page 133 for more information on how the linker interprets input.

You can also use wildcard characters to specify multiple object files. For example, use *.OBJ to specify all the object files in a directory.

Using response files

Instead of specifying linker input on the command line, you can put options and filename parameters in a response file. You can combine the response file with options and parameters on the command line.

When you invoke the linker, use the following syntax:

```
ilink @responsefile
```

The value for *responsefile* is the name of the response file. The @ symbol indicates that the file is a response file. If the file is not in the working directory, specify the path for the file as well as the file name.

You can begin using a response file at any point on the linker command line. Although multiple response files can be specified on the command line, they cannot be nested.

Options can appear anywhere in the response file. If an option is not valid, the linker generates an error message and stops linking.

Specify the contents of the response file just as you would on the command line. Because the default syntax identifies input by file extension rather than by position on

the command line, it does not matter how many lines there are, or whether there are blank lines in the file.

Example

The response file named FUN.LNK contains the following:

```
/DEBUG /MAP
fun.obj text.obj table.obj care.obj
/exec
/map:funlist
graf.lib
```

When you enter `ilink @fun.lnk`, the linker does the following:

- Links the four object modules `fun.obj`, `text.obj`, `table.obj`, and `care.obj` into an .EXE file named `fun.exe`. Because no output type is specified, the linker defaults to .exe.
- Generates the map file `funlist.map` (assuming the extension `.map`).
- Preserves debugging information (because of the `/DEBUG` option).
- Links any needed routines from the library file `graf.lib`, and from the default PL/I libraries specified in the object files.

Specifying executable output type

You can use the linker to produce executable modules (.EXE) and dynamic link libraries (.DLL). The linker produces .EXE files by default.

Use options to specify what kind of output you want:

- To produce an .EXE, specify the `/EXEC` option. OS/2 Or, include the module statement NAME.
- To produce a .DLL, specify the `/DLL` option. OS/2 Or, include the module statement LIBRARY.

Producing an .EXE file

The linker produces .EXE files by default. Use the `/EXEC` option, OS/2 or the `NAME` module statement, to explicitly identify the output file as an .EXE file.

An .EXE file is one that can be executed directly. You can run the program by typing the name of the file. In contrast, DLL and device driver programs execute when they are called by other processes, and cannot be run independently.
To reduce the size of the .EXE file and improve its performance, use the following options:

- OS/2 /ALIGNMENT:*value* to set the alignment factor in the output file. Set *value* to smaller factors to reduce the size of the executable, and to larger factors to reduce load time for the executable. By default, the alignment is set to `512`.

Specifying executable output type

- /BASE:0x00010000 to specify the load address for the executable. The load address must be `0x00010000`. Any other value produces a warning, and is not used. Specifying this value explicitly allows the linker to omit relocation records, which can result in a smaller executable.
- /EXEPACK to compress the file.
- WIN /ALIGNFILE:*n* to set the file alignment for sections in the output file. Set *n* to smaller factors to reduce the size of the executable, and to larger factors to reduce load time for the executable. By default, the alignment is set to `512`.
- /BASE:*n* to specify the load address for the executable. For example, if several DLLs are loaded at base addresses that ensure that the DLLs do not overlap, the linker does not have to reapply the relocation records. `n` (the load address) must be a multiple of 0x10000, and it cannot be 0.

If you do not specify an extension for the output file name, the linker automatically adds the extension .EXE to the name you provide. If you do not specify an output filename at all, the linker generates an .EXE file with the same filename as the first .OBJ file it linked.

OS/2 If you are using a module definition (.DEF) file, you can include a NAME statement to provide a name for the application.

Producing a dynamic link library

A dynamic link library (.DLL) file contains executable code for common functions, just as a library (.LIB) file does. When you link with a DLL (using an import library), the code in the DLL is **not** copied into the executable file. Instead, only the import definitions for DLL functions are copied, resulting in a smaller executable. At run time, the dynamic link library is loaded into memory, along with the .EXE file.

To produce a DLL as output, compile at least one object file with the DLLINIT compiler option, and link it with the `/DLL` linker option. OS/2 If you are using a module definition (.DEF) file, specify the LIBRARY statement in the .DEF file. WIN You must include an export definition (.EXP) file that specifies which functions are to be included in the DLL.

You can find more information in Chapter 21, “Building dynamic link libraries” on page 386.

To reduce the size of the DLL and improve its performance, use the following options:

- OS/2 /ALIGNMENT:value to set the alignment factor in the output file. Set *value* to smaller factors to reduce the size of the DLL, and to larger factors to reduce load time for the DLL. By default, the alignment is set to `512`.
- /EXEPACK to compress the file.
- WIN /ALIGNFILE:value to set the alignment factor in the output file. Set *value* to smaller factors to reduce the size of the DLL, and to larger factors to reduce load time for the DLL. By default, the alignment is set to `512`.

For DLLs, setting a /BASE value can save load time when the given load address is available. If the load address is not available, the /BASE value is ignored, and there is no load time benefit.

OS/2 If you use the /BASE option, set /BASE:0x12000000 (or a lesser value), and provide a separate value for each DLL.

Once you have produced the DLL, you can produce an executable that links to the DLL.

OS/2 The linker determines which functions your object files need during the linking process. This process can be done in two ways:

Using a .DEF file
Provide a .DEF file when you create the executable. In the .DEF file, use the IMPORTS statement to specify which of the DLL's functions your object files need.

Using an import library.
Use the ILIB utility to create an import library. When you use an import library, you no longer need to use the IMPORTS statement. The linker determines which functions your object files need during the linking process.

WIN Use the ILIB utility to create an import library, and then use the `.LIB` file as input to the linker.

Packing executables

OS/2 Specify the `/EXEPACK` linker option to reduce the size of the executable by compressing pages in the file. The operating system automatically decompresses the pages when the program is loaded.

Specify `/PACKCODE` to produce slightly faster and more compact code by grouping neighboring code segments that have similar attributes.

Specify `/PACKDATA` to produce more compact files by grouping neighboring data segments that have similar attributes.

Specify `/DBGPACK` when you are debugging, to reduce the size of the executable file and potentially improve debugger performance.

Generating a map file

Specify /MAP to generate a map file, which lists the object modules in your output file; section names, addresses, and sizes; and symbol information. If you do not specify a name for the map file, the map file takes the name of the executable output file, with the extension .MAP. To prevent the map file from being generated, use the default, /NOMAP.

Specify /LINENUMBERS to include source file line numbers and associated addresses in the map file.

Linker return codes

The linker has the following return codes:

Code	Meaning
0	The link was completed successfully. The linker detected no errors, and issued no warnings.
4	**Warnings** issued. There may be problems with the output file.
8	**Errors** detected. The linking might have completed, but the output file cannot be run successfully.
12	Both warnings issued and errors detected (see return codes 4 and 8)
16	**Severe errors** detected. Linking ended abnormally, and the output file cannot be run successfully.
20	Both warnings issued and severe errors detected (see return codes 4 and 16)
24	Both errors and severe errors issued (see return codes 8 and 16)
28	The linker issued warnings, detected errors, and detected severe errors (see return codes 4, 8, and 16)

If you invoke the linker through a makefile, you can force NMAKE to ignore warnings by putting `-7` before the `ILINK` command.

Chapter 8. Setting linker options

Setting linker options

Linker options are not case sensitive, so you can specify them in lower-, upper-, or mixed case. You can also substitute a dash (-) for the slash (/) preceding the option. For example, -DEBUG is equivalent to /DEBUG. You can specify options in either a short or long form. For example, /DE, /DEB, and /DEBU are all equivalent to /DEBUG. Lower- and uppercase, short and long forms, dashes, and slashes can all be used on one command line, as in:

```
ilink /de -DBGPACK -Map /NOI prog.obj
```

Separate options with a space or tab character. You can specify linker options in the following ways:

- On the command line
- In the ILINK environment variable
- Using WorkFrame.

Options specified on the command line override the options in the ILINK environment variable.

Some linker options take numeric arguments. You can enter numbers in decimal, octal, or hexadecimal format. See “Specifying numeric arguments” on page 146 for more information.

Setting options on the command line

Linker options specified on the command line override any previously specified in the ILINK environment variable (as described in “Setting options in the ILINK environment variable”).

You can specify options anywhere on the command line. Separate options with a space or tab character.

For example, to link an object file with the /MAP option, enter:

```
ilink /M myprog.obj
```

Setting options in the ILINK environment variable

Store frequently used options in the ILINK environment variable. This method is useful if you find yourself repeating the same command-line options every time you link. You cannot specify file names in the environment variable, only linker options.

The ILINK environment variable can be set either from the command line, in a command (.CMD) file, or in the CONFIG.SYS file. If it is set on the command line or by running a command file, the options will only be in effect for the current session (until you reboot your computer). If it is set in the CONFIG.SYS file, the options are set when you boot your computer, and are in effect every time you use the linker unless you override them using a .CMD file or by specifying options on the command line.

In the following example, options on the command line override options in the environment variable. If you enter the following commands:

Specifying numeric arguments

```
SET ILINK=/NOI /AL:256 /DE
ILINK test
ILINK /NODEF /NODEB prog
```

The first command sets the environment variable to the options /NOIGNORECASE, /ALIGNMENT:256, and /DEBUG

The second command links the file `test.obj`, using the options specified in the environment variable, to produce `test.exe`

The last command links the file `prog.obj` to produce `prog.exe`, using the option /NODEFAULTLIBRARYSEARCH, in addition to the options /NOIGNORECASE and /ALIGNMENT:256. The /NODEBUG option on the command line overrides the /DEBUG option in the environment variable, and the linker links without the /DEBUG option.

Specifying numeric arguments

Some linker options and module statements take numeric arguments. You can specify numbers in any of the following forms:

Decimal Any number **not** prefixed with 0 or 0x is a decimal number. For example, 1234 is a decimal number.

Octal Any number prefixed with 0 (but not 0x) is an octal number. For example, 01234 is an octal number.

Hexadecimal Any number prefixed with 0x is a hexadecimal number. For example, 0x1234 is a hexadecimal number.

Summary of OS/2 linker options

Table 10. OS/2 linker options summary

Option	Description	Default
"/?"	Display help	None
"/ALIGNMENT"	Set alignment factor	/A:512
"/BASE, /NOBASE"	Set preferred loading address	/BAS:0x00010000
"/CODEVIEW, NOCODEVIEW"	Include debugging information	/NOC
"/DBGPACK, /NODBGPACK"	Pack debugging information	/NODB
"/DEBUG, /NODEBUG"	Include debugging information	/NODEB
"/DEFAULTLIBRARYSEARCH"	Search default libraries	/DEF
"/DLL"	Generate DLL	/EXEC
"/EXEC"	Generate .EXE file	/EXEC
"/EXEPACK, /NOEXEPACK"	Compress data	/NOEXE
"/EXTDICTIONARY, /NOEXTDICTIONARY"	Use extended dictionary to search libraries	/EXT
"/FORCE"	Create executable output file even if errors	/NOFO
"/FREEFORMAT, /NOFREEFORMAT"	Use free format command line syntax	/FR
"/HELP"	Display help	None
"/IGNORECASE, /NOIGNORECASE"	Ignore capitalization in identifiers	/NOI
"/INFORMATION, /NOINFORMATION"	Display status of linking process	/NOIN
"/LINENUMBERS, /NOLINENUMBERS"	Include line numbers in map file	/NOLI
"/LOGO, /NOLOGO"	Display logo, echo response file	/LO
"/MAP, /NOMAP"	Generate map file	/NOM
"/OPTFUNC, /NOOPTFUNC"	Remove unreferenced functions	/NOOPTF
"/OUT"	Name output file	Name of first .OBJ file
"/PACKCODE, /NOPACKCODE"	Pack neighboring code segments with similar attributes	/PACKC: 0xFfffFfff
"/PACKDATA, /NOPACKDATA"	Pack neighboring data segments with similar attributes	/PACKD: 0xFfffFfff
"/PMTYPE"	Specify application type	None
"/SECTION"	Set attributes for segment	Accept default attributes
"/SEGMENTS"	Set maximum number of segments	/SE:128
"/STACK"	Set stack size of application	/ST:32768

Linker options for OS/2

OS/2 ⇨ This section describes the linker options in alphabetical order.

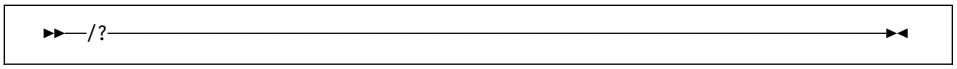
For each option, the description includes:

- The syntax for specifying the option
- The default setting

OS/2 linker options

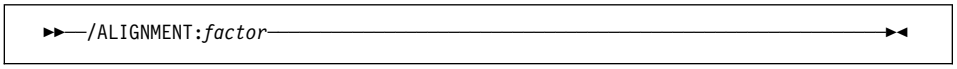
- Any accepted abbreviations
- A description of the option and its parameters, and any interaction it may have with other options.

/?



OS/2 Use /? to display a list of valid linker options. This option is equivalent to /HELP.

/ALIGNMENT



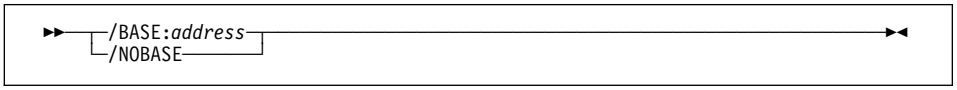
OS/2 Use /ALIGNMENT to set the alignment factor in the .EXE or .DLL file.

The alignment factor determines where pages in the .EXE or .DLL file start. From the beginning of the file, the start of each page is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 1 to 4096.

Default: /ALIGNMENT:512

Abbreviation: /A

/BASE, /NOBASE



OS/2 Use /BASE to specify the preferred load address for the first load segment of a .DLL file. The number you specify in *address* is rounded up to the nearest multiple of 64K. The second load segment is then loaded at the next available multiple of 64K, and so on.

If the file load segments cannot be loaded beginning at this preferred address, then the preferred address is ignored and the objects are loaded according to the internal relocation records retained in the file data.

For .EXE files, use the default base address of 64K (/BASE:0x00010000). Specifying this address explicitly can slightly reduce the size of the executable. Any other address result in a warning, and 64K is used.

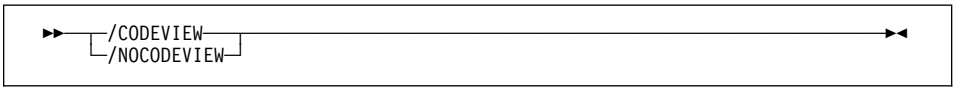
This option has the same effect as the BASE module definition file statement. If you specify both the BASE statement and the /BASE option, the statement value overrides the option value.

Specify /NOBASE to retain relocation records and emit internal fixups, when you generate an .EXE file. This does not affect the actual base address, or interfere with any value you specified with /BASE. You can specify both options.

Default: /BASE:0x00010000

Abbreviations: /BAS

/CODEVIEW, NOCODEVIEW



OS/2 These options will not be available in future releases of the linker. Use /DEBUG, /NODEBUG instead.

Use /CODEVIEW to include debug information in the output file, so you can debug the file with the debugger, or trace its execution with the Performance Analyzer. The linker embeds symbolic data and line number information in the output file.

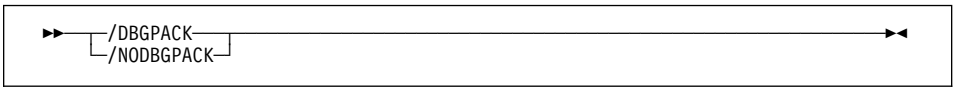
For debugging, compile the object files with TEST. For the Performance Analyzer, compile the object files with PROFILE and GONUMBER.

/CODEVIEW provides the same functionality as /DEBUG and is provided only for purposes of compatibility. Linking with /CODEVIEW or /DEBUG increases the size of the executable output file.

Default: /NOCODEVIEW

Abbreviations: /C|/NOC

/DBGPACK, /NODBGPACK



OS/2 Use /DBGPACK to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.

Performance Consideration: Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can actually speed up your linking, because there is less information to write to file.

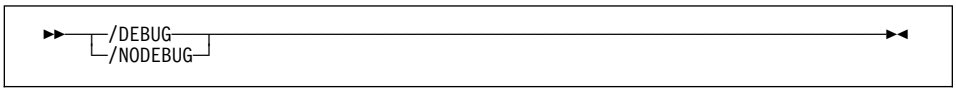
You can only pack debug information in objects created with version 3.0 of the compiler or later. If you use /DBGPACK with older object files, the linker generates a warning and does not pack the debug information.

When you specify /DBGPACK, /DEBUG is turned on by default.

Default: /NODBGPACK

Abbreviations: /DB|/NODB

/DEBUG, /NODEBUG



OS/2 Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with the performance analyzer. The linker will embed symbolic data and line number information in the output file.

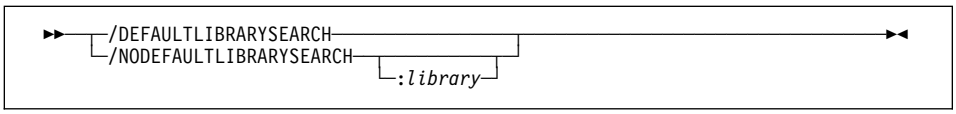
For debugging, compile the object files with TEST.. For the Performance Analyzer, compile the object files with PROFILE and GONUMBER.

Note: Linking with /DEBUG increases the size of the executable output file.

Default: /NODEBUG

Abbreviations: /DE|/NODEB

/DEFAULTLIBRARYSEARCH



OS/2 Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

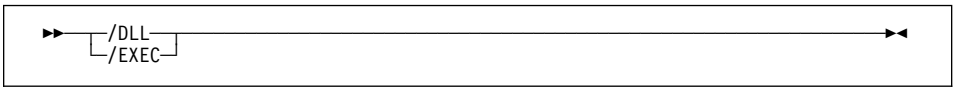
Use /NODEFAULTLIBRARYSEARCH to tell the linker to ignore default libraries when it resolves external references. If you specify a *library* with the option, the linker ignores that default library, but searches any others that are defined in the object files.

If you specify /NODEFAULTLIBRARYSEARCH, then you must explicitly specify all the libraries you want to use, including VA PL/I runtime libraries and any OS/2 libraries you need.

Default: /DEFAULTLIBRARYSEARCH

Abbreviations: /DEF|/NOD

/DLL



OS/2 Use /DLL to identify the output file as a dynamic link library (.DLL file). You can also identify the output file as a DLL with the LIBRARY statement in a module definition file.

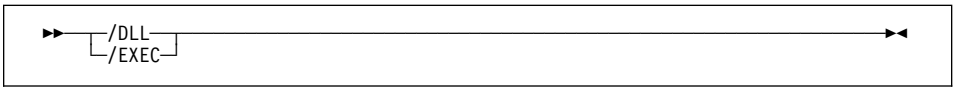
If you specify /DLL with /EXEC, then only the last specified of the options takes effect.

If you do not specify /DLL, then by default the linker produces an .EXE file (/EXEC).

Default: /EXEC

Abbreviations: None

/EXEC



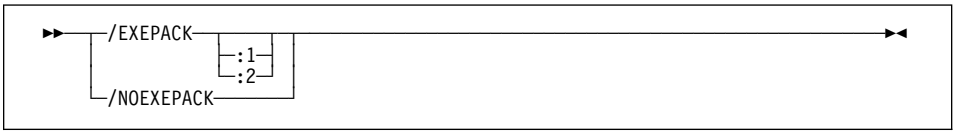
OS/2 Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default. You can also identify the output as an .EXE file with the NAME statement in a module definition file. If you specify /EXEC with /DLL, only the last specified of the options takes effect.

If you do not specify /EXEC, the linker produces an .EXE file by default.

Default: /EXEC

Abbreviations: None

/EXEPACK, /NOEXEPACK



OS/2 Use /EXEPACK to reduce the size of the executable by compressing pages in the file. The operating system automatically decompresses the pages when the program runs.

Specify /EXEPACK[:1] to compress data segments in your output file, using run-length encoding compression. If compression does not reduce the size of the segment, the linker does not compress that segment.

Specify /EXEPACK:2 to compress both data and code segments, as follows:

- For data segments, the linker tries both LZW compression and run-length encoding compression, and uses the method with the more efficient result.
- For code segments, the linker uses LZW compression.

Segments are evaluated one page at a time. If compression does not reduce the size of the page, the page is not compressed.

OS/2 V3.0 only: Only set /EXEPACK:2 if you are developing for OS/2 version 3.0 or later. OS/2 version 2.1 or earlier cannot run programs that have been compressed with /EXEPACK:2.

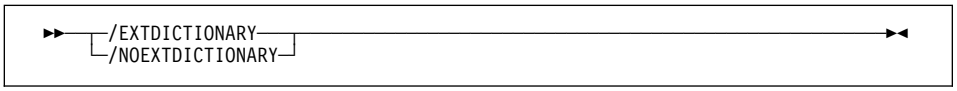
Linking and compressing generally takes longer than linking alone, because of the extra time spent compressing. However, if the compression is effective enough, it can actually speed up the linking process, because there is less information to write to file.

By default, the linker does not compress the output file.

Default: /NOEXEPACK

Abbreviations: /E|/NOEXE

/EXTDICTIONARY, /NOEXTDICTIONARY



OS/2 Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see if that module requires other modules in the library, and then pulls in the additional modules automatically.

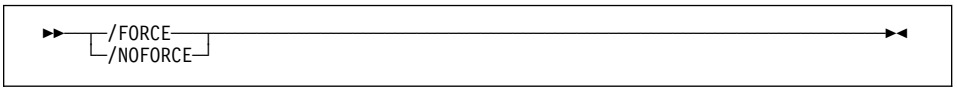
The linker searches the extended dictionary by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are defining a symbol in your object code that is also defined in one of the libraries to which you are linking. Otherwise the linker issues error L2044 because you have defined the same symbol in two different places. When you link with /NOEXTDICTIONARY, the linker searches the dictionary directly, instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

Default: /NOEXTDICTIONARY

Abbreviations: /EXT|/NOE

/FORCE



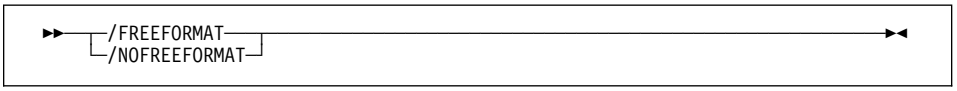
OS/2 Use /FORCE to produce an executable output file even if there are errors during the linking process.

By default, the linker does not produce an executable output file if it encounters an error.

Default: /NOFORCE

Abbreviations: /FO|/NOFO

/FREEFORMAT, /NOFREEFORMAT



OS/2 Use /FREEFORMAT to allow free placement of files, options, and directories on the command line, separated by space or tab characters. Use the /OUT option to name the executable output file. Use the /MAP option to name the map file. Library and definition files are identified by their extension.

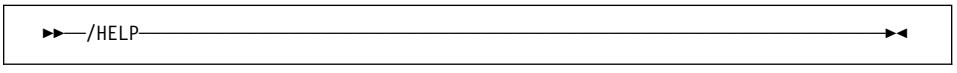
/FREEFORMAT is in effect by default.

Use /NOFREEFORMAT to allow a LINK386-compatible command line syntax, in which different types of file are grouped and separated by commas. If you specify /NOFREEFORMAT, then you cannot specify /OUT. Instead, specify a name for the executable output file in the appropriate place in the command line syntax.

Default: /FREEFORMAT

Abbreviations: /FR|/NOFR

/HELP



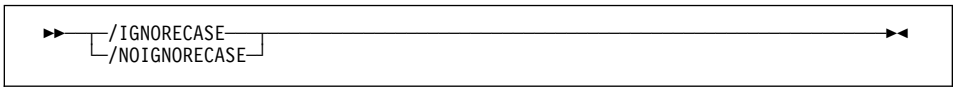
OS/2 Use /HELP to display a list of valid linker options. This option is equivalent to /?.

Default: None

Abbreviation: /H

OS/2 linker options

/IGNORECASE, /NOIGNORECASE



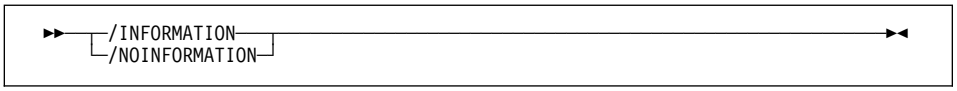
OS/2 Use /IGNORECASE to turn off case sensitivity, ignoring capitalization in identifiers. For example, with this option on, the linker treats ABC, abc, and Abc as equivalent.

By default, the linker is case sensitive, and would treat ABC, abc, and Abc as unique names.

Default: /NOIGNORECASE

Abbreviations: /IG|/NOI

/INFORMATION, /NOINFORMATION



OS/2 Use /INFORMATION to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

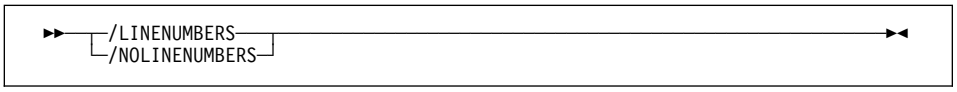
If you are having trouble linking because the linker is finding the wrong files or finding them in the wrong order, use /INFORMATION to determine the locations of the object files being linked and the order in which they are linked.

The output from this option is sent to **stdout**. You can redirect the output to a file using OS/2 redirection symbols.

Default: /NOINFORMATION

Abbreviations: /I|/NOIN

/LINENUMBERS, /NOLINENUMBERS



OS/2 Use /LINENUMBERS to include source file line numbers and associated addresses in the map file. For this option to take effect, there must already be line number information in the object files you are linking.

When you compile, use the GONUMBER option to include line numbers in the object file (or the TEST option to include all debugging information).

If you give the linker an object file without line number information, the /LINENUMBERS option has no effect.

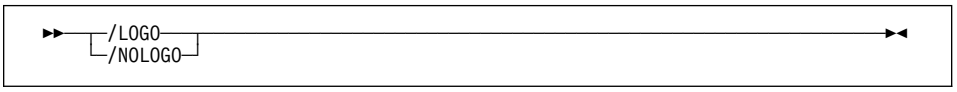
The /LINENUMBERS option forces the linker to create a map file, even if you specified /NOMAP.

By default, the map file is given the same name as the output file, plus the extension .map. You can override the default name by specifying a map file name.

Default: /NOLINENUMBERS

Abbreviations: /L|/NOLI

/LOGO, /NOLOGO



OS/2 Use /NOLOGO to suppress the product information that appears when the linker starts. /NOLOGO also stops the contents of the response file from being echoed to the screen.

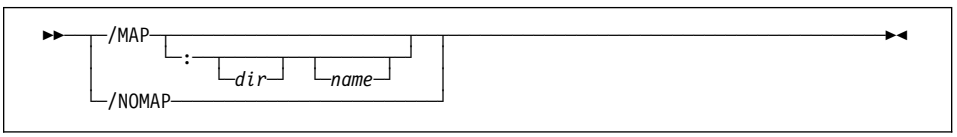
Specify /NOLOGO before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

By default, the linker displays product information at the start of the linking process, and displays the contents of the response file as it reads the file.

Default: /LOGO

Abbreviations: /LO|/NOL

/MAP, /NOMAP



OS/2 Use /MAP to generate a map file with the name *name*, and in the directory *dir*, that lists the composition of each segment, and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name, and in order of address.

If you do not specify *dir*, the map file is generated into the current working directory. If you do not specify *name*, the map file has the same name as the executable output file, with the extension .map.

OS/2 linker options

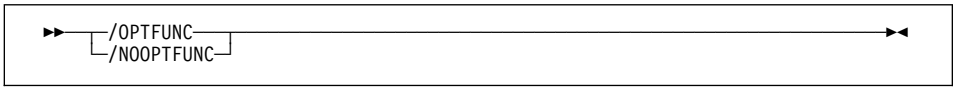
For compatibility with LINK386, you can specify /MAP:full. With the VA PL/I linker, this is the same as specifying /MAP.

Note: If you are linking with the /NOFREE option, you can specify a name for the map file in the *map* parameter. Any name you specify with the /MAP option is ignored.

Default: /NOMAP

Abbreviations: /M|/NOM

/OPTFUNC, /NOOPTFUNC



OS/2 Use /OPTFUNC to remove unreachable functions. The linker removes functions that are:

- Not referenced anywhere in the object code
- Rendered unreferenced by the removal of other functions
- Not exported for use in other files

When the function is removed, any additional functions that were required only by that function are also removed. Removing the functions and code reduces the size of your .EXE or .DLL output file.

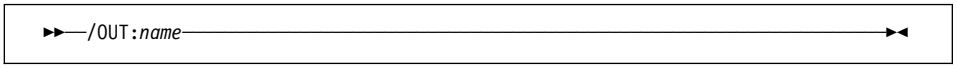
By default, the linker does not remove unreachable functions.

Performance Consideration: Optimized linking generally takes longer than regular linking, because of the extra processing that the linker performs. However, if the optimization is effective enough, it can actually speed up the linking process, because there is less information to write to file. Generally, you may want to link without the /OPTFUNC option, until your code is tested and stable.

Default: /NOOPTFUNC

Abbreviations: /OPTF|/NOOPTF

/OUT



OS/2 Use /OUT to specify a name for the executable output file. To use /OUT, you must be using the default command line syntax (/FREEFORMAT). If you are using the /NOFREE (LINK386-compatible) format, then you cannot use the /OUT option.

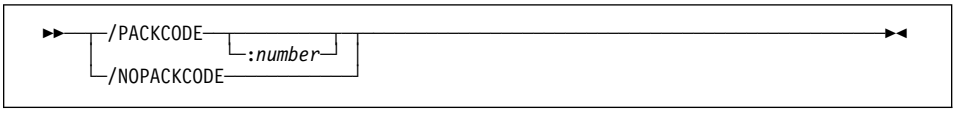
If you do not provide an extension with *name*, then the linker provides an extension based on the type of file you are producing:

File produced	Default extension
Executable program	.EXE
Dynamic link library	.DLL

Default: Name of first .OBJ file with appropriate extension.

Abbreviation: /O

/PACKCODE, /NOPACKCODE



OS/2 Use /PACKCODE to produce slightly faster and more compact code. The linker groups neighboring code segments that have similar attributes, and assigns them to the same load segment. The linker adjusts offsets to each routine upward as required.

Specify *number* to set the maximum size for a load segment. The linker will start new load segments as necessary to avoid exceeding the maximum.

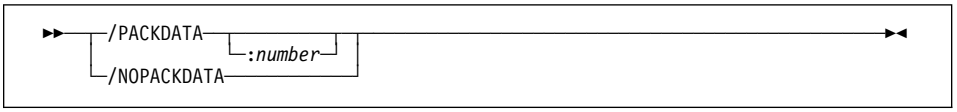
For 16-bit segments, *number* is ignored, and 65500 is used instead. Use /NOPACKCODE to turn off code segment packing.

Use the /OPTFUNC option to reduce the size of your output files even further.

Default: /PACKCODE:0xFfffFfff

Abbreviations: /PACKC|/NOP

/PACKDATA, /NOPACKDATA



OS/2 Use /PACKDATA to produce more compact files by grouping neighboring data segments that have similar attributes, and assigning them to the same load segment.

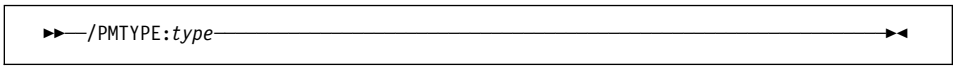
Specify *number* to set the maximum size for a load segment. The linker will start new load segments as necessary to avoid exceeding the maximum. By default, the linker sets a maximum of 0xFfffFfff.

Default: /NOPACKDATA

Abbreviations: /PACKD|/NOPACKD

OS/2 linker options

/PMTYPE



OS/2 Use /PMTYPE to specify the type of .EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs). The option is equivalent to the NAME module statement, but uses different type names.

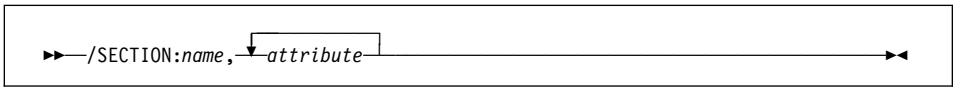
Table 11. /PMTYPE Parameters

Type	Description	Equivalent NAME Statement Parameter
PM	Presentation Manager application. The application uses the API provided by the Presentation Manager, and must run in the Presentation Manager environment.	WINDOWAPI
VIO	Application compatible with Presentation Manager. The application can run inside the Presentation Manager, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager applications.	WINDOWCOMPAT
NOVIO	Application that is not compatible with the Presentation Manager and must run in a separate screen group from the Presentation Manager.	NOTWINDOWCOMPAT

Default: None

Abbreviation: /PM

/SECTION



OS/2 Use /SECTION to specify memory-protection attributes for the *name* segment. You can specify the following attributes:

Letter	Sets Attribute
E	EXECUTE
R	READ
S	SHARED
W	WRITE

The following example sets the READ and SHARED attributes, but not the EXECUTE, or WRITE attributes, for the segment dseg1 in an .EXE file:

```
/SEC:dseg1,RS
```

Defaults

Segments are assigned attributes by default, as follows:

Segment	Default Attributes
Code segments	EXECUTE, READ (ER) Correspond to the SEGMENTS attribute EXECUTEREAD.
Data segments (in .EXE file)	READ, WRITE (RW) Correspond to the SEGMENTS attribute READWRITE.
Data segments (in .DLL file)	READ, WRITE, SHARED (RWS) Correspond to the SEGMENTS attributes READWRITE and SHARED.
CONST32_RO segment	READ, SHARED (RS) Correspond to the SEGMENTS attributes READONLY and SHARED.

You can also set these attributes, and other attributes, to segments using statements in a module definition file:

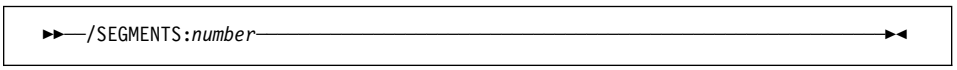
CODE	Sets default attributes for CODE segments
DATA	Sets default attributes for DATA segments
SEGMENTS	Sets attributes for specific segments

Assignments given in a module definition file override any assignments made with /SECTION.

Default: Depends on segment type

Abbreviation: /SEC

/SEGMENTS



OS/2 Use /SEGMENTS to set the number of logical segments a program can have. You can set *number* to any value in the range 1 to 16375. See “Specifying numeric arguments” on page 146.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

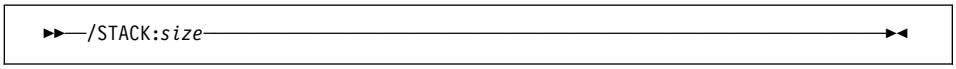
OS/2 linker options

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting *number* to the actual number of segments in the program.

Default: /SEGMENTS:256

Abbreviation: /SE

/STACK



OS/2 Use /STACK to set the stack size (in bytes) of your program. The size must be an even number from 0 to 0xFFFFFFFE. If you specify an odd number, it is rounded up to the next even number.

You cannot specify a stack size in which the second most significant byte is either 02 or 04 (in hex), because of a restriction in OS/2 2.0. The linker issues a warning, and adds 64k to the specified stack size to avoid this restriction.

For example, if you specify /STACK:0x00020000 the linker adds 64k, which results in /STACK:0x00030000

Similarly, if you specify /STACK:0x11041111 the linker adds 64k, which results in /STACK:0x11051111

If your program generates a stack-overflow message, use /STACK to increase the size of the stack. If your program uses very little stack space, you can save space by decreasing the stack size. If the executable is a visual application containing more than 10 windows, you should add about 10K to the stack size for each additional window.

If your program uses a visual part containing more than 10 windows, then add about another 10K to the stack size for each additional window in that part. For example, if the most windows contained in any one part is 18, then specify /ST:1134688 (that is, (1024 × 10 × 8) + 32768).

Note: Once the executable is produced, you can still change its stack size, using the EXEHDR utility in the Warp toolkit.

/STACK is equivalent to the STACKSIZE statement in a module definition (.DEF) file. If you specify both the statement and the option, the statement value overrides the option value.

Default: /STACK:32768 (32K)

Abbreviation: /ST

Summary of Windows linker options

Table 12. Windows linker options summary

Option	Description	Default
"/?"	Display help	None
"/ALIGNADDR"	Set address alignment	/A:0x00010000
"/ALIGNFILE"	Set file alignment	/A:512
"/BASE"	Set preferred loading address	/BAS:0x00400000
"/CODE"	Set section attributes for executable	/CODE:RX
"/DATA"	Set section attributes for data	/DATA:RW
"/DBGPACK, /NODBGPACK"	Pack debugging information	/NODB
"/DEBUG, /NODEBUG"	Include debugging information	/NODEB
"/DEFAULTLIBRARYSEARCH"	Search default libraries	/DEF
"/DLL"	Generate DLL	/EXEC
"/DLL"	Specify an entry point in an executable file	None
"/EXECUTABLE"	Generate .EXE file	/EXEC
"/EXTDICTIONARY, /NOEXTDICTIONARY"	Use extended dictionary to search libraries	/EXT
"/EXTDICTIONARY, /NOEXTDICTIONARY"	Do not relocate the file in memory	/NOFI
"/FORCE"	Create executable output file even if errors are detected	/NOFO
"/HEAP"	Set the size of the progam heap	/HEAP:0x100000,0x1000
"/HELP"	Display help	None
"/INCLUDE"	Forces a reference to a symbol	None
"/INFORMATION, /NOINFORMATION"	Display status of linking process	/NOIN
"/LINENUMBERS, /NOLINENUMBERS"	Include line numbers in map file	/NOLI
"/LOGO, /NOLOGO"	Display logo, echo response file	/LO
"/MAP, /NOMAP"	Generate map file	/NOM
"/OUT"	Name output file	Name of first .obj file
"/PMTYPE"	Specify application type	/PMTYPE:VIO
"/SECTION"	Set attributes for section	Set by `/CODE` and `/DATA`
"/SEGMENTS"	Set maximum number of segments	/SE:256
"/STACK"	Set stack size of application	/STACK: 0x100000,0x1000
"/STUB"	Specify the name of the DOS stub file	None
"/SUBSYSTEM"	Specify the required subsystem and version	/SUBSYSTEM: WINDOWS,4.0
"/VERBOSE"	Display status of linking process	/NOV
"/VERSION"	Write a version number in the run file	/VERSION:0.0

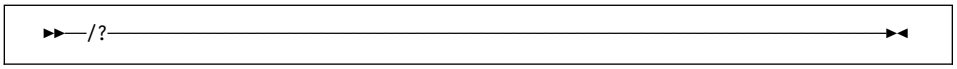
Windows linker options

This section describes the linker options in alphabetical order.

For each option, the description includes:

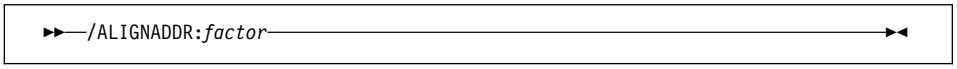
- The syntax for specifying the option.
- The default setting.
- Any accepted abbreviations.
- A description of the option and its parameters, and any interaction it may have with other options.

/?



WIN Use /? to display a list of valid linker options. This option is equivalent to /HELP.

/ALIGNADDR



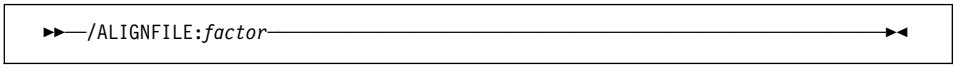
WIN Use /ALIGNADDR to set the address alignment for segments.

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 256M.

Default: /ALIGNADDR:0x00010000

Abbreviation: /ALIGN

/ALIGNFILE



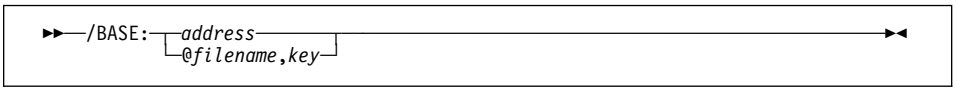
WIN Use /ALIGNFILE to set the file alignment for segments.

The alignment factor determines where segments in the .EXE or .DLL file start. From the beginning of the file, the start of each segment is aligned at a multiple (in bytes) of the alignment factor. The alignment factor must be a power of 2, from 512 to 64K.

Default: /ALIGNFILE:512

Abbreviation: /A

/BASE



WIN Use /BASE to specify the preferred load address for the first load segment of a .DLL file.

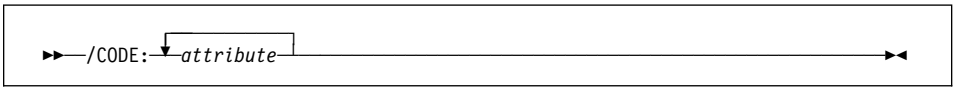
Specifying *@filename, key*, in place of *address*, bases a set of programs (usually a set of DLLs) so they do not overlap in memory. *filename* is the name of a text file that defines the memory map for a set of files. *key* is a reference to a line in *filename* beginning with the specified key. Each line in the memory-map file has the syntax: *key address maxsize*

Separate the elements with one or more spaces or tabs. The *key* is a unique name in the file. The *address* is the location of the memory image in the virtual address space. The *maxsize* is an amount of memory within which the image must fit. The linker will issue a warning when the memory image of the program exceeds the specified size. A comment in the memory-map file begins with a semicolon (;) and runs to the end of the line.

Default: /BASE:0x00400000

Abbreviations: /BAS

/CODE



WIN Use /CODE to specify the default attributes for all code sections. Letters can be specified in any order.

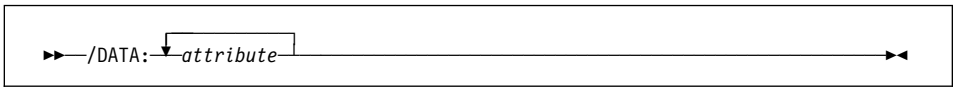
Letter	Attribute
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

Default: /CODE:RX

CODE description abbreviations: None

Windows linker options

/DATA



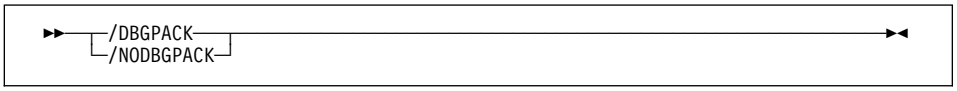
WIN Use /DATA to specify the default attributes for all data sections. Letters can be specified in any order.

Letter	Attribute
E or X	EXECUTE
R	READ
S	SHARED
W	WRITE

Default: /DATA:RW

Abbreviations: None

/DBGPACK, /NODBGPACK



WIN Use /DBGPACK to eliminate redundant debug type information. The linker takes the debug type information from all object files and needed library components, and reduces the information to one entry per type. This results in a smaller executable output file, and can improve debugger performance.

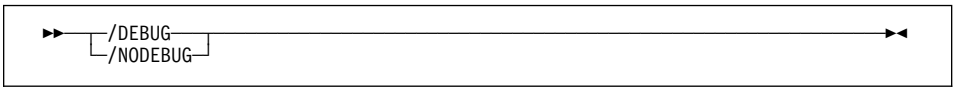
Performance Consideration: Generally, linking with /DBGPACK slows the linking process, because it takes time to pack the information. However, if there is enough redundant debug type information, /DBGPACK can actually speed up your linking, because there is less information to write to file.

When you specify /DBGPACK, /DEBUG is turned on by default.

Default: /NODBGPACK

Abbreviations: /DB|/NODB

/DEBUG, /NODEBUG



WIN Use /DEBUG to include debug information in the output file, so you can debug the file with the debugger, or analyze its performance with Performance Analyzer. The linker will embed symbolic data and line number information in the output file.

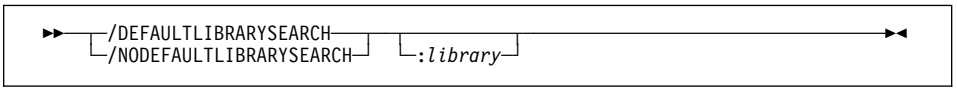
For debugging, compile the object files with TEST.

For the Performance Analyzer, compile the object files with PROFILE and GONUMBER. Linking with /DEBUG increases the size of the executable output file.

Default: /NODEBUG

Abbreviations: /D|/NODEB

/DEFAULTLIBRARYSEARCH



WIN Use /DEFAULTLIBRARYSEARCH to have the linker search the default libraries of object files when resolving references.

If you specify a *library* with the option, the linker adds the library name to the list of default libraries. The default libraries for an object file are defined at compile time, and embedded in the object file. The linker searches the default libraries by default.

Use /NODEFAULTLIBRARYSEARCH to tell the linker to ignore default libraries when it resolves external references. If you specify a *library* with the option, the linker ignores that default library, but searches the rest of the default libraries (and any others that are defined in the object files).

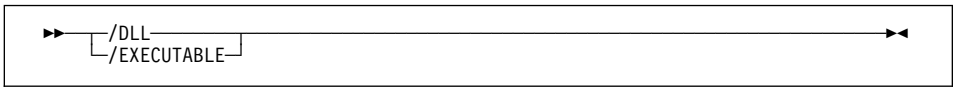
If you specify /NODEFAULTLIBRARYSEARCH without specifying *library*, then you must explicitly specify all the libraries you want to use, including VA PL/I runtime libraries.

Default: /DEFAULTLIBRARYSEARCH

Abbreviations: /DEF|/NOD

Windows linker options

/DLL



WIN Use /DLL to identify the output file as a dynamic link library (.DLL file). The object files should be compiled with the PL/I option DLLINIT.

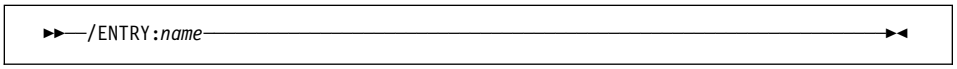
If you specify /DLL with /EXEC, only the last specified of the options takes effect.

If you do not specify /DLL, or any of the other options, then by default the linker produces an .EXE file (/EXEC).

Default: /EXECUTABLE

Abbreviation: /EXEC

/ENTRY

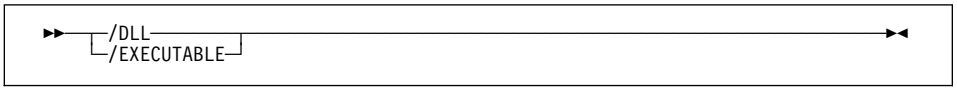


WIN Use /ENTRY to specify an entry point (name of a routine or function) in an executable.

Default: None

Abbreviation: /EN

/EXECUTABLE



WIN Use /EXEC to identify the output file as an executable program (.EXE file). The linker generates .EXE files by default.

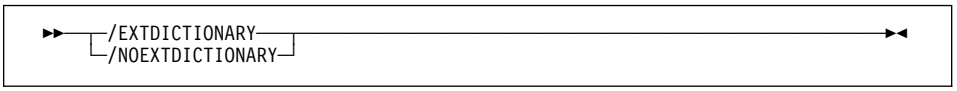
If you specify /EXEC with /DLL, only the last specified of the options takes effect.

If you do not specify /EXEC or /DLL, then by default the linker produces an .EXE file.

Default: /EXECUTABLE

Abbreviation: /EXEC

/EXTDICTIONARY, /NOEXTDICTIONARY



WIN Use /EXTDICTIONARY to have the linker search the extended dictionaries of libraries when it resolves external references. The extended dictionary is a list of module relationships within a library. When the linker pulls in a module from the library, it checks the extended dictionary to see if that module requires other modules in the library, and then pulls in the additional modules automatically.

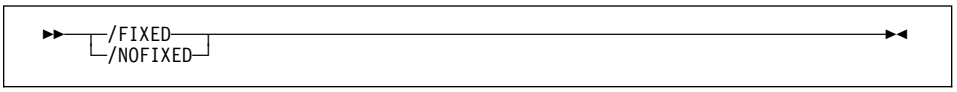
The linker searches the extended dictionary by default, to speed up the linking process.

Use /NOEXTDICTIONARY if you are defining a symbol in your object code that is also defined in one of the libraries to which you are linking. Otherwise the linker issues an error because you have defined the same symbol in two different places. When you link with /NOEXTDICTIONARY, the linker searches the dictionary directly, instead of searching the extended dictionary. This results in slower linking, because references must be resolved individually.

Default: /EXTDICTIONARY

Abbreviations: /EXT|/NOE

/FIXED, /NOFIXED



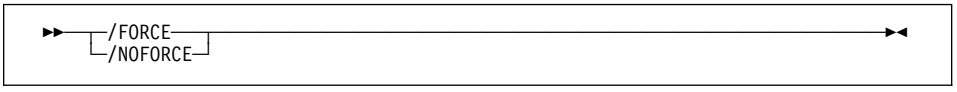
WIN Use /FIXED to tell the loader not to relocate a file in memory when the specified base address is not available.

For more information on base addresses, see the /BASE linker option.

Default: /NOFIXED

Abbreviations: /FI|/NOFI

/FORCE



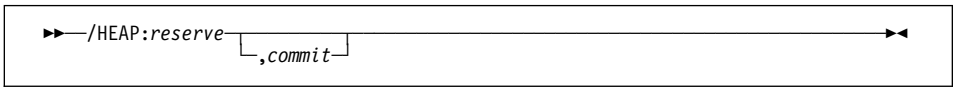
WIN Use /FORCE to produce an executable output file even if there are errors during the linking process.

Default: /NOFORCE

Windows linker options

Abbreviations: /FO|/NOFO

/HEAP

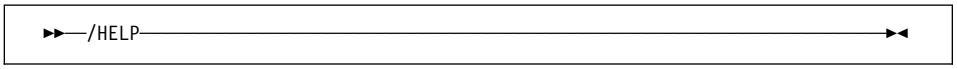


WIN Use /HEAP to set the size of the program heap in bytes. The *reserve* argument sets the total virtual address space reserved. The *commit* sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, but execution time can be slower.

Default: /HEAP:0x100000,0x1000

Abbreviation: /HEA

/HELP

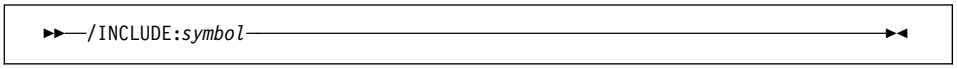


WIN Use /HELP to display a list of valid linker options. This option is equivalent to /?.

Default: None

Abbreviation: /H

/INCLUDE

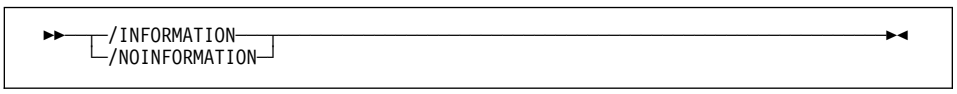


WIN Use /INCLUDE to force a reference to a symbol. The linker searches for an object module that defines the symbol.

Default: None

Abbreviation: /INC

/INFORMATION, /NOINFORMATION

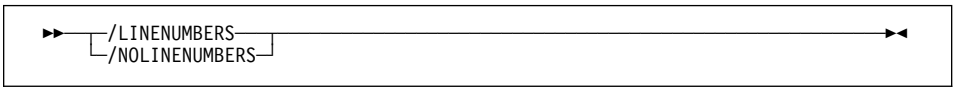


WIN See the description of the /VERBOSE linker option.

Default: /NOINFORMATION

Abbreviations: /I|/NOIN

/LINENUMBERS, /NOLINENUMBERS



WIN Use /LINENUMBERS to include source file line numbers and associated addresses in the map file. For this option to take effect, there must already be line number information in the object files you are linking.

When you compile, use the GONUMBER option to include line numbers in the object file (or the TEST option to include all debugging information).

If you give the linker an object file without line number information, the /LINENUMBERS option has no effect.

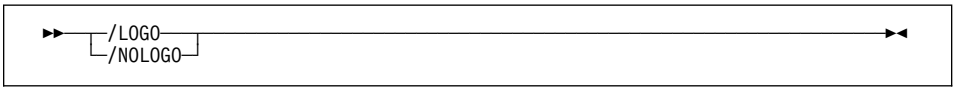
The /LINENUMBERS option forces the linker to create a map file, even if you specified /NOMAP.

By default, the map file is given the same name as the output file, plus the extension .map. You can override the default name by specifying a map filename.

Default: /NOLINENUMBERS

Abbreviations: /L|/NOLI

/LOGO, /NOLOGO



WIN Use /NOLOGO to suppress the product information that appears when the linker starts.

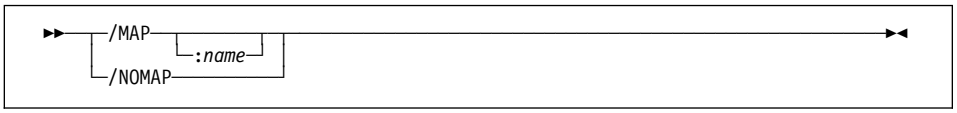
Specify /NOLOGO before the response file on the command line, or in the ILINK environment variable. If the option appears in or after the response file, it is ignored.

Default: /LOGO

Abbreviations: /LO|/NOL

Windows linker options

/MAP, /NOMAP



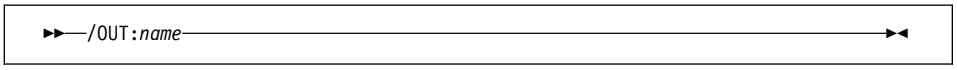
WIN Use /MAP to generate a map file called *name*. The file lists the composition of each segment, and the public (global) symbols defined in the object files. The symbols are listed twice: in order of name and in order of address.

If you do not specify a directory, the map file is generated into the current working directory. If you do not specify *name*, the map file has the same name as the executable output file, with the extension **.**map.

Default: /NOMAP

Abbreviations: /M|/NOM

/OUT



WIN Use /OUT to specify a name for the executable output file.

If you do not provide an extension with *name*, then the linker provides an extension based on the type of file you are producing:

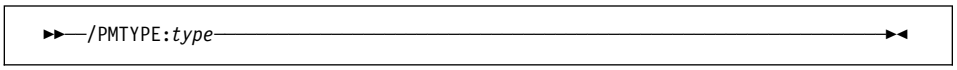
File produced	Default extension
Executable program	.EXE
Dynamic link library	.DLL

If you do not use the /OUT option, then the linker uses the filename of the first object file you specified, with the appropriate extension.

Default: Name of first .OBJ file with appropriate extension.

Abbreviation: /O

/PMTYPE



WIN Use /PMTYPE to specify the type of .EXE file that the linker generates. Do not use this option when generating dynamic link libraries (DLLs).

One of the following types must be specified:

PM The executable must be run in a window.

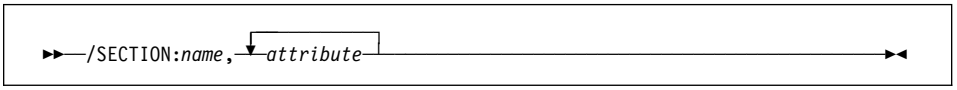
VIO The executable can be run either in a window or in a full screen.

NOVIO The executable must not be run in a window; it must use a full screen.

Default: /PMTYPE:VIO

Abbreviation: /PM

/SECTION



WIN Use /SECTION to specify memory-protection attributes for the *name* section. *name* is case sensitive. You can specify the following attributes:

Letter	Sets Attribute
E or **X**	EXECUTE
R	READ
S	SHARED
W	WRITE

The following example sets the READ and SHARED attributes, but not the EXECUTE, or WRITE attributes, for the section dseg1 in an .EXE file.

```
/SEC:dseg1,RS
```

Defaults

Sections are assigned attributes by default, as follows:

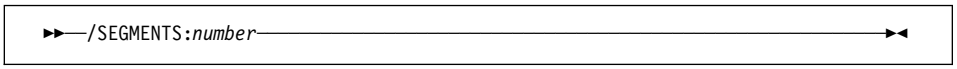
Segment	Default Attributes
Code sections	EXECUTE, READ (ER)
Data sections (in .EXE file)	READ, WRITE (RW), not shared
Data sections (in .DLL file)	READ, WRITE, not shared
CONST32_RO section	READ, SHARED (RS)

Default: Depends on segment type

Abbreviation: /SEC

Windows linker options

/SEGMENTS



WIN Use /SEGMENTS to set the number of logical segments a program can have. You can set *number* to any value in the range 1 to 16375. See "Specifying numeric arguments" on page 146.

For each logical segment, the linker must allocate space to keep track of segment information. By using a relatively low segment limit as a default (256), the linker is able to link faster and allocate less storage space.

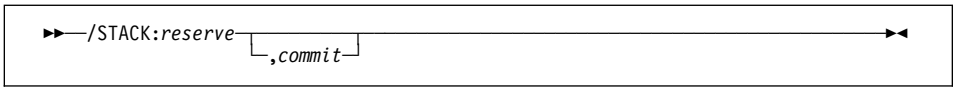
When you set the segment limit higher than 256, the linker allocates more space for segment information. This results in slower linking, but allows you to link programs with a large number of segments.

For programs with fewer than 256 segments, you can improve link time and reduce linker storage requirements by setting *number* to the actual number of segments in the program.

Default: /SEGMENTS:256

Abbreviation: /SE

/STACK



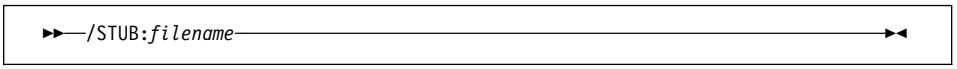
WIN Use /STACK to set the stack size (in bytes) of your program. The size must be an even number from 0 to 0xFfffFffe. If you specify an odd number, it is rounded up to the next even number.

reserve indicates the total virtual address space reserved. *commit* sets the amount of physical memory to allocate initially. When *commit* is less than *reserve*, memory demands are reduced, although execution time may be slower.

Default: /STACK:0x100000,0x1000

Abbreviation: /ST

/STUB

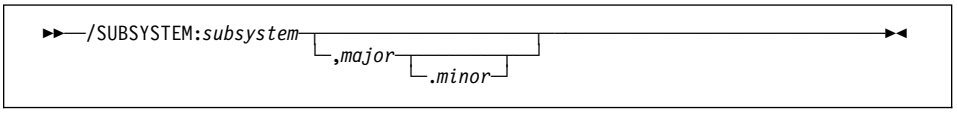


WIN Use /STUB to specify the name of the DOS executable at the beginning of the output file created.

Default: None

Abbreviation: /STU

/SUBSYSTEM



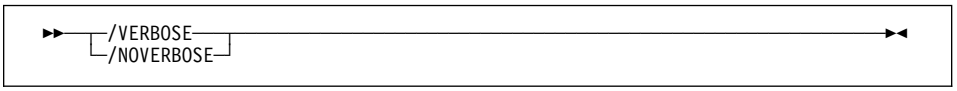
WIN Use /SUBSYSTEM to specify the subsystem and version required to run the program. The *major* and *minor* arguments are optional and specify the minimum required version of the subsystem. The *major* and *minor* arguments are integers in the range 0 to 65535.

Subsystem	Major.Minor	Description
WINDOWS	3.10	A graphical application that uses the Graphical Device Interface (GDI) API.
CONSOLE	3.10	A character-mode application that uses the Console API.

Default: /SUBSYSTEM:WINDOWS,4.0

Abbreviation: /SU

/VERBOSE



WIN Use /VERBOSE to have the linker display information about the linking process as it occurs, including the phase of linking and the names and paths of the object files being linked.

If you are having trouble linking because the linker is finding the wrong files or finding them in the wrong order, use /VERBOSE to determine the locations of the object files being linked and the order in which they are linked.

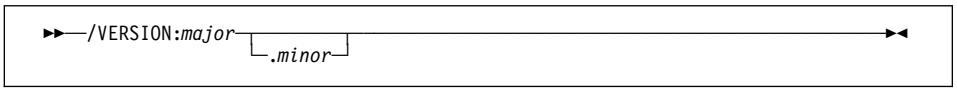
The output from this option is sent to **stdout**. You can redirect the output to a file using Windows redirection symbols.

/VERBOSE is the same as /INFORMATION.

Default: /NOVERBOSE

Abbreviations: /VERB|/NOV

/VERSION



WIN Use /VERSION to write a version number in the header of the run file. The *major* and *minor* arguments are integers in the range 0 to 65535.

Default: /VERSION:0.0

Abbreviation: /VER

Part 3. Running and debugging your program

Chapter 9. Using run-time options

Once you have prepared the executable form of your PL/I program, you need to test its execution behavior. The first step is to run the program and see what happens. Depending on the nature of your application, you might need to do some input and output setup (SET statements) before invoking the program.

Setting run-time environment variables

You can set the run-time environment for your program by using environment variables.

PATH

Use the PATH environment variable to specify the search path for EXE and CMD files not in the current directory.

```
set path=c:\ibm;d:\project
```

OS/2 If you set the PATH variable in your CONFIG.SYS file, you must use the SET command.

You can specify one or more directories with this variable. Given the preceding example, the current directory is searched first, followed by `c:\ibm` and then `d:\project`.

DPATH

Use DPATH to specify the search path for run-time messages. The program searches for them first in the current directory, then in the directory or directories specified by the DPATH variable. The following example would cause the program to search the current directory followed by `c:\set1` and `d:\set2` in that order.

```
set dpath=c:\set1;d:\set2
```

OS/2 To set the DPATH variable in your CONFIG.SYS file, you must use the SET command.

LIBPATH (OS/2)

OS/2 Use LIBPATH to specify the search path for DLL files required by a program. The library DLLs and any user DLLs must be in one of the directories specified by the LIBPATH.

This variable can only be specified in the CONFIG.SYS file. The following example sets the DLL search path to the `c:\cmlib`, `c:\ibm\dll`, and `c:\ibm\lib` directories:

```
libpath=c:\cmlib;c:\ibm\dll;c:\ibm\lib
```

LIBPATH cannot be specified using the SET command. For more information on DLLs, see Chapter 21, “Building dynamic link libraries” on page 386.

STEPLIB (OS/2)

OS/2 You can use STEPLIB to specify a DLL search path applicable to FETCHed DLLs only. The path specified by STEPLIB is searched **before** the path specified by LIBPATH in the CONFIG.SYS file. The use of STEPLIB is optional.

```
set steplib=c:\dlls1;c:\dlls2
```

If you set the STEPLIB variable in your CONFIG.SYS file, you must use the SET command.

Specifying run-time options

Each time your application executes, a set of run-time options is established. These options determine some of the properties of the application's execution, such as allocation of storage and production of reports. IBM supplies defaults for each of the run-time options; however, you can change them as needed prior to running your application.

Where to specify run-time options

You can alter the default settings for run-time options in an environment variable and in the application source code. Alternatives, from lowest priority to highest priority, are:

- **Using the IBM defaults**
- **Setting run-time options in the CEE.OPTIONS environment variable**

  Use the SET command in your CONFIG.SYS (or AUTOEXEC.BAT) file or on the command line to specify run-time options by means of the CEE.OPTIONS environment variable. For example:

  ```
  set cee.options=natlang(enu)
  ```

  As mentioned above, there are two methods for setting options in the CEE.OPTIONS environment variable. The first method, setting CEE.OPTIONS in CONFIG.SYS (or AUTOEXEC.BAT) has lower priority than the second, using the SET command.

  1. Setting CEE.OPTIONS in CONFIG.SYS (or AUTOEXEC.BAT)

     Run-time options specified in CONFIG.SYS (or AUTOEXEC.BAT) are the options in effect for every session you start. This is a good place to specify run-time options that you want to have in effect for every application you run.

     If CEE.OPTIONS already exists in the CONFIG.SYS (or AUTOEXEC.BAT) file, change or add to the existing variable.

2. Run-time options specified in a SET command on the command line are in effect only for that session or window and override any run-time options specified in CONFIG.SYS (or AUTOEXEC.BAT). This is the recommended method.

   To change run-time option settings, use a SET command with the desired settings. Each SET command completely replaces any previous SET commands, including a SET command in CONFIG.SYS (or AUTOEXEC.BAT). Therefore, you must include the settings of all run-time options from any previous SET command if you still want them in effect in each subsequent SET command.

   For example, assume you have the following in your CONFIG.SYS (or AUTOEXEC.BAT) file:

   ```
   set cee.options=natlang(jpn)
   ```

   and later enter this command from the command line:

   ```
   set cee.options=natlang(enu)
   ```

   This means that NATLANG has returned to its default value, which is NATLANG(ENU).

   To return all run-time options to the IBM-supplied defaults, set CEE.OPTIONS to a null argument:

   ```
   set cee.options=
   ```

   With OS/2 and Windows, you can group several commands, including a SET command for CEE.OPTIONS, in a command file. Running such a command file is equivalent to issuing each of the commands individually on the command line.

Specifying multiple run-time options or suboptions

When specifying a string of run-time options, you must separate each option with a comma without any embedded spaces.

Use commas to separate suboptions of run-time options. If you do not specify a suboption, you must still specify the comma to indicate its omission. Trailing commas are not required. If you do not specify any suboptions, the defaults are used. For example, NATLANG() is valid syntax.

Default settings for the options are indicated in the options syntax diagrams or in the descriptions of suboptions, where applicable.

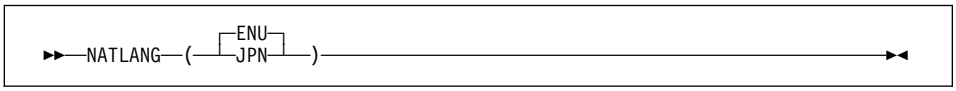
Run-time options

This section describes the run-time option NATLANG.

NATLANG

NATLANG

The NATLANG option specifies the national language to be used for run-time messages. Message translations are provided for Japanese and mixed-case U.S. English. NATLANG also determines how the message facility formats messages.



JPN
This is a 3-character id specifying Japanese. Message text can be a mixture of SBCS (single-byte character set) and DBCS (double-byte character set) characters.

ENU
This is a 3-character id specifying mixed-case U.S. English. Message text is made up of SBCS characters and consists of both upper and lowercase letters.

Default: NATLANG(ENU)

Run-time option and storage reports, as well as dump output, are written only in mixed-case U.S. English.

If you specify a national language that is unavailable on your system, the default is used.

Shipping run-time DLLs

If you are shipping DLLs with your application, this list should help you determine which ones you need based on your application.

OS/2 For OS/2, the following files are needed for a non-multithreading application:

- DLL\HEPOS20.DLL
- DLL\IBMOS20.DLL
- DLL\IBMOSTB.DLL
- DLL\IBMOS20F.DLL
- DLL\IBMOS20G.DLL
- MSG\IBMRTENU.MSG

These files are needed for multithreading applications on OS/2:

- DLL\HEPOM20.DLL
- DLL\IBMOM20.DLL
- DLL\IBMOMTB.DLL
- DLL\IBMOM20F.DLL
- DLL\IBMOM20G.DLL
- MSG\IBMRTENU.MSG

Shipping run-time DLLs

In addition to those listed previously, if your OS/2 application uses BTRIEVE, you must also ship DLL\IBMPBTRV.DLL. If PLISRTx is used in an application, you must include DLL\IBMSRTX.DLL. For a VSAM application, add the following:

- DLL\DUBRUN.DLL
- DLL\DUBLDM.DLL

WIN For Windows, the following files are needed for a non-multithreading application:

- BIN\HEPWS20.DLL
- BIN\IBMWS20.DLL
- BIN\IBMWSTB.DLL
- BIN\IBMWS20F.DLL
- BIN\IBMWS20G.DLL
- BIN\IBMRTENU.DLL

These files are needed for multithreading applications on Windows:

- BIN\HEPWM20.DLL
- BIN\IBMWM20.DLL
- BIN\IBMWMTB.DLL
- BIN\IBMWM20F.DLL
- BIN\IBMWM20G.DLL
- BIN\IBMRTENU.DLL

In addition to those listed previously, if your Windows application uses BTRIEVE, you must also ship BIN\IBMPBTRV.DLL.

Your application containing a copy of any of these files or modules must be labeled as follows:

CONTAINS
IBM VisualAge PL/I Version 2.1
Runtime Modules

Chapter 10. Testing and debugging your programs

Effective design and coding practices help you create quality programs and should be followed by thorough testing of those programs. You should give adequate attention to the testing phase of development so that:

- Your program becomes fully operational after the fewest possible test runs, thereby minimizing the time and cost of program development.
- Your program is proven to have fulfilled all of its design objectives before it is released for production work.
- Your program contains sufficient comments to enable those who use and maintain the program to do so without additional assistance.

The process of testing usually uncovers *bugs*, a generic term that encompasses anything that your program does that you did not expect it to do. The process of removing these bugs from your program is called *debugging*.

While this chapter does not attempt to provide an exhaustive coverage of testing and debugging, it does provide useful tips and techniques to help you produce top-quality, error-free PL/I programs. Both general and PL/I-specific testing and debugging information follow.

Testing your programs

Testing your PL/I programs can be difficult, especially if the programs are logically complex or involve numerous modules. Do not skip this step, though, because it is important to detect and remove bugs from a program before it moves into a production environment.

Here are three testing approaches that you can apply to all of your PL/I programs:

Code inspection

Also called desk checking, code inspection involves selecting a piece of code and reading it from the viewpoint of the computer. With either a printed copy of the source program or an online view of the source file, follow the flow of the program. Where there is input data, guess at some likely data and substitute it for variable values. When there is a calculation, do the calculation manually or with a calculator, and so on. Code inspection often reveals logic problems, syntax errors, and bugs that the compiler misses (for example, “n + 2” instead of “n*2”).

Data testing

You provide a program with test data to verify that it runs as designed. The purpose of data testing is to see if the program takes exception (for example, a run-time error) to any possible data that it might have to handle in a production environment. Therefore, you need to use a wide variety of data to test your program.

For example, have your program process extremes of data that you know lead to errors (such as the OVERFLOW condition) and see how the program responds. Your program should incorporate error checking (such as ERROR ON-units) to accommodate any possible data.

General debugging tips

Attention: You should never test with irreplaceable data, nor should you store irreplaceable data within access of a program being tested!

Path testing

The data that you use for testing a program should be selected to test all parts of the program. In other words, if your program consists of a number of modules, the data that you test the program with should require the use of all of the modules. If your program can take five possible paths at a given point, you should provide sets of data that take the program down each of the five paths.

As your program becomes more and more complex, providing the program with data to accommodate every possible path combination might become practically impossible. However, it is important that you select test cases that check a representative range of paths. For example, rather than check every possible iteration of a DO-loop, test the first, last, and one intermediate case.

Bugs are discovered as you test your programs and removing those bugs sometimes requires being able to reproduce them. Therefore, when you test programs, always begin from a known state. For example, when a bug is encountered you should know the values of variables, the compile-time options used, the contents of memory, and so on. PL/I provides features such as SNAP and PLIDUMP that help you do this.

As a rule, a program that ran perfectly well yesterday but reveals a bug today does so because of one or more changes to the state of the machine. Therefore, when testing your PL/I programs be sure to know, in detail, the state of the machine at compile time and at run time.

General debugging tips

Debugging is a process of letting your program run until it does something that you did not expect it to do. After finding a bug, you modify the program so that it does not encounter the bug when the program is in the exact machine state that initially produced the bug. This is accomplished by a combination of back-tracking, intuition, and trial and error. The major obstacle to effective debugging is that removing one bug can introduce new bugs into your program. You should consider general debugging tips as well as some debugging techniques specific to PL/I.

Consider the following tips when debugging your programs:

Make one change at a time

When attempting to remedy a bug, introduce only one change into the source code of your program at a time. By introducing a single change, you can compare the program behavior before and after the change to accurately measure the effect of the change.

Follow program logic sequence

Fix your program's bugs in the order in which they are encountered when the program is run.

Watch for unexpected results
Locate a given bug in the program source code at a point that corresponds to an unexpected change in the state of program execution.

For example, the undesired change in the state of program execution might be the unintended assignment of the decimal value “100” to the character variable “z.” In this case, you might find that the source code has an error that assigns the wrong variable in an assignment statement.

PL/I debugging techniques

PL/I provides you with a number of methods for program debugging which are described in the following sections:

Compile-time options
Footprints
Dumps
Error and condition handling

Using compile-time options for debugging

The PL/I workstation products are designed to diagnose many of the bugs in your programs at compile time, and provides you with a compiler listing that explains what mistakes you made and where you made them. In addition, you can use compile-time options to make the compiler listing even more useful (for additional information on using compiler listings, see “Compile-time options summary” on page 27).

The following compile-time options are useful for debugging your PL/I programs:

FLAG
Suppresses the listing of diagnostic messages below a certain severity and terminates compilation if a specified number of messages is reached. If your program is not behaving as expected and the compiler messages do not explain the problem, you might want to use FLAG to include informational messages in the compiler listing. These messages (otherwise suppressed by default) might help explain problems in your program. For additional information on using FLAG, see “FLAG” on page 51.

GONUMBER
Creates a statement number table that is needed for debugging.

PREFIX
Enables or disables specified PL/I conditions. Because you can specify the conditions with a compile-time option, you do not need to change your source program. Compiling with PREFIX(SUBRG STRZ STRG) can be very helpful in debugging. For more information on using PREFIX, see “PREFIX” on page 69.

RULES
Specifies the strictness with which various language rules are enforced by the compiler. You can use it to flag common programming errors.

You might find the following suboptions for RULES particularly useful for debugging:

NOLAXIF
Disallows IF, WHILE, UNTIL, and WHEN clauses to evaluate to other than BIT(1) NONVARYING.

NOLAXDCL
Disallows all implicit and contextual declarations except for built-ins and the files SYSIN and SYSPRINT.

NOLAXQUAL
The compiler flags any reference to structure members that are not level 1 and are not dot qualified.

For example, consider the program:

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

With RULES(NOLAXDCL) in effect, the two typographical errors above are considered implicit declarations by the compiler and are flagged as errors. For more information on using RULES, see “RULES” on page 73.

SNAP
Specifies that the compiler produces a listing of trace information that is useful for locating errors in your program.

For detailed information on using SNAP for debugging, see “SNAP dumps for trace information” on page 192.

For more information on SNAP syntax, see “SNAP” on page 77.

XREF
Specifies that the compiler listing includes a table of names used in the program together with the numbers of the statements in which they are referenced or set. This allows you to easily track where names are used in your source program. For more information on using XREF, see “XREF” on page 85.

Using footprints for debugging

When debugging, it is useful to periodically check:

- Where your program is in its execution flow (for example, which module is being run).
- The value of identifiers so that you can see when they change and what values they are assigned.

To accomplish these tasks, you can use built-in functions, PUT DATA and PUT LIST statements, and display statements. These approaches are described in more detail in the following sections.

Built-in functions

The built-in functions PROCNAME, PACKAGENAME, and SOURCELINE are useful in following the execution of your program when you are trying to track the location of a problem and the sequence of events that caused it. The following statement can be inserted wherever you want to display the procedure name and line number of the statement currently being executed.

```
display (procname() || sourceline());
```

PUT LIST

Allows you to transmit strings and data items to the data stream (for example, to a printer-destined output file). For example, the following procedure lets you know if the FIXEDOVERFLOW condition is raised, and prints out the value of the variable that led to the condition (in this case, z):

```
Debug: Proc(x);
   dcl x fixed bin(31);
   on fixedoverflow
      begin;
         put skip list('Fixedoverflow raised because z = '||z);
      end;
   end;
   get list(z);
   x = 8 * z;
```

If z is too large, multiplying it by 8 produces a value that is too large for any FIXED BIN(31) variable and would therefore raise the FIXEDOVERFLOW condition. PUT SKIP LIST transmits the data (in this case, the string "`Fixedoverflow raised because z = ...`") to the default file SYSPRINT. You can define SYSPRINT using export DD= statements. For more information on using SYSPRINT, see "Using SYSIN and SYSPRINT files" on page 245.

PUT DATA

Allows you to transmit the value of data items to the output stream. For example, if you specified the following line in your program, it would transmit the values of `string1` and `string2` to the output stream (for example, to SYSPRINT):

```
put data (string1, string2);
```

DISPLAY

You can use DISPLAY to transmit information to your monitor. This can be useful to let you know how far a program has progressed, what procedure a program is running, and so on. For example:

```
Display ('End of job!');
Display ('Reached the MATH procedure');
Display ('Hurrah! Got past the string manipulation stuff...');
```

Using DISPLAY with PUT statements results in output appearing in unpredictable order. For more information on using the DISPLAY statement, see “DISPLAY statement input and output” on page 230.

Using dumps for debugging

When you are debugging your programs, it is often useful to obtain a printout (a dump) of all or part of the storage used by your program. You can also use a dump to provide trace information. Trace information helps you locate the sources of errors in your program.

Two types of dumps are useful:

PLIDUMP
SNAP

Use of the IMPRECISE compile-time option might lead to incomplete trace information. For additional information on the IMPRECISE option, see “IMPRECISE” on page 52.

Formatted PL/I dumps—PLIDUMP

You use PLIDUMP to obtain:

- Trace information that allows you to locate the point-of-origin of a condition in your source program.
- File information, including: the attributes of the files open at the time of the dump, the values of certain file-handling built-in functions, and the contents of the I/O storage buffer.

To get a formatted PL/I dump, you must include a call to PLIDUMP in your program. The statement CALL PLIDUMP can appear wherever a CALL statement appears. It has the following form:

```
call plidump('dump options string', 'dump title string');
```

dump options string
An expression specifying a string consisting of any of the following dump option characters:

T–Trace
PL/I generates a calling trace.

NT–No trace
The dump does not give a calling trace.

F–File information
The dump gives a complete set of attributes for all open files, plus the contents of all accessible I/O buffers.

NF–No file information
The dump does not give file information.

S–Stop
The program ends after the dump.

E–Exit
The current thread or the program (if it is the main thread) ends after the dump.

K Ignored.

NK Ignored.

C–Continue
The program continues after the dump.

PL/I reads options from left to right. It ignores invalid options and, if contradictory options exist, takes the rightmost options.

dump title string
An expression that is converted to character if necessary and printed as a header on the dump. The string has no practical length limit. PL/I prints this string as a header to the dump. If the character string is omitted, PL/I does not print a header.

If the program calls PLIDUMP a number of times, the program should use a different user-identifier character string on each occasion. This simplifies identifying the point at which each dump occurs. In addition to this header, each new invocation of PLIDUMP prints another heading above the user-identifier showing the date, time, and page number 1.

PLIDUMP defaults: The default dump options are T, F, and C with a null dump title string:

```
plidump('TFC', ' ');
```

Suggested PLIDUMP coding: A program can call PLIDUMP from anywhere in the program, but the normal method of debugging is to call PLIDUMP from an ON-unit. Because continuation after the dump is optional, the program can use PLIDUMP to get a series of dumps while the program is running.

You can use the *DD:plidump* environment variable to specify where the PLIDUMP output should be located, for example:

```
set dd:plidump = d:\mydump;
```

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

PLIDUMP example: When you run the program shown in Figure 5 on page 190, a formatted dump is produced as shown in Figure 6 on page 191.

```
TestDump: proc options(main);
   declare
      Sysin input file,
      Sysprint stream print file;
   open file(Sysprint);
   open file(Sysin);
   put skip list('AbCdEfGhIjKlMnOpQrStUvWxYz');
   call IssueDump;

   IssueDump: proc;
      call plidump( ' ', 'Testing PLIDUMP');
   end IssueDump;

end TestDump;
```

Figure 5. PL/I code that produces a formatted dump

The call to PLIDUMP in the IssueDump procedure does not specify any PLIDUMP options (they appear as the first of the two character strings), so the defaults are used. Also note that the PL/I default files SYSIN and SYSPRINT have been explicitly opened so that the formatted dump displays the contents of their portions of the I/O buffer.

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

2  User identifier: Testing PLIDUMP


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

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

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

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

Figure 6. Example of PLIDUMP output

1 Time and date when PLIDUMP is called. Each separate PLIDUMP call has this information.

2 Character string specified in the PLIDUMP call (the second of the two strings provided to PLIDUMP) that is useful in helping to identify the dump if a number of dumps are produced.

3 Trace information, delineated by `* * * Calling trace * * *` and `* * * End of calling trace * * *`. This information allows you to trace back through the procedures from which PLIDUMP was called. In the example above, PLIDUMP was called from the procedure ISSUEDUMP which is nested in the TESTDUMP procedure. The hexadecimal offsets of each procedure are also provided in the trace information.

The trace information is provided by default as the T option and can be suppressed by specifying the NT option for PLIDUMP.

4 File attributes of SYSIN (opened explicitly in the program).

5 ENVIRONMENT options for the file SYSIN.

6 Values of relevant I/O built-in functions for the file SYSIN.

7 Contents of the I/O buffer for the SYSIN file. The first column is the hexadecimal address, the following columns are the hexadecimal contents of memory.

8 Contents of the I/O buffer for SYSPRINT. Notice that the second character string supplied to PLIDUMP (`AbCd...`) is contained in the I/O buffer, as seen by the text representation of the I/O buffer at the right-hand side of the row.

SNAP dumps for trace information

While not a "dump" in the strictest sense, the SNAP compile-time option is used to find out what error conditions are raised in your program and where they are raised. SNAP provides the same trace information provided by PLIDUMP "T" option (see "Formatted PL/I dumps—PLIDUMP" on page 188). Like PLIDUMP, SNAP can be issued multiple times throughout one run of a program.

An example of a call for a SNAP dump is:

```
on attention snap;
```

This statement calls for a SNAP dump if the ATTENTION condition is raised.

Using error and condition handling for debugging

PL/I condition handling is a powerful tool for debugging programs. All errors detected at run-time are associated with conditions. You can handle these conditions in one of the following ways:

- Writing ON-units that specify what your program should do if a given condition is raised
- Accepting the standard system action

Error and condition handling terminology

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

Established
An ON-unit becomes established when the ON statement is executed. It ceases to be established when an ON or REVERT statement referring to the same condition is executed, or when the associated block is terminated.

Enabled
A condition is enabled when the occurrence of the condition results in the execution of an ON-unit or standard action.

Interrupts and PL/I conditions

Certain PL/I conditions are detected by machine interrupts. Others have to be detected by special testing code either in the run-time library modules or in the compiled program.

Statically and dynamically descendant

Static and dynamic descendant are terms used to define the scope of error-handling features. ON-units are dynamically descendant; that is, they are inherited from the calling procedure in all circumstances. Condition enablement is statically descendant; that is, it is inherited from the containing block in the source program. Statically descendant procedures can be determined during compilation. Dynamically descendant procedures might not be known until run-time. Figure 7 shows an example of statically and dynamically descendant procedures.

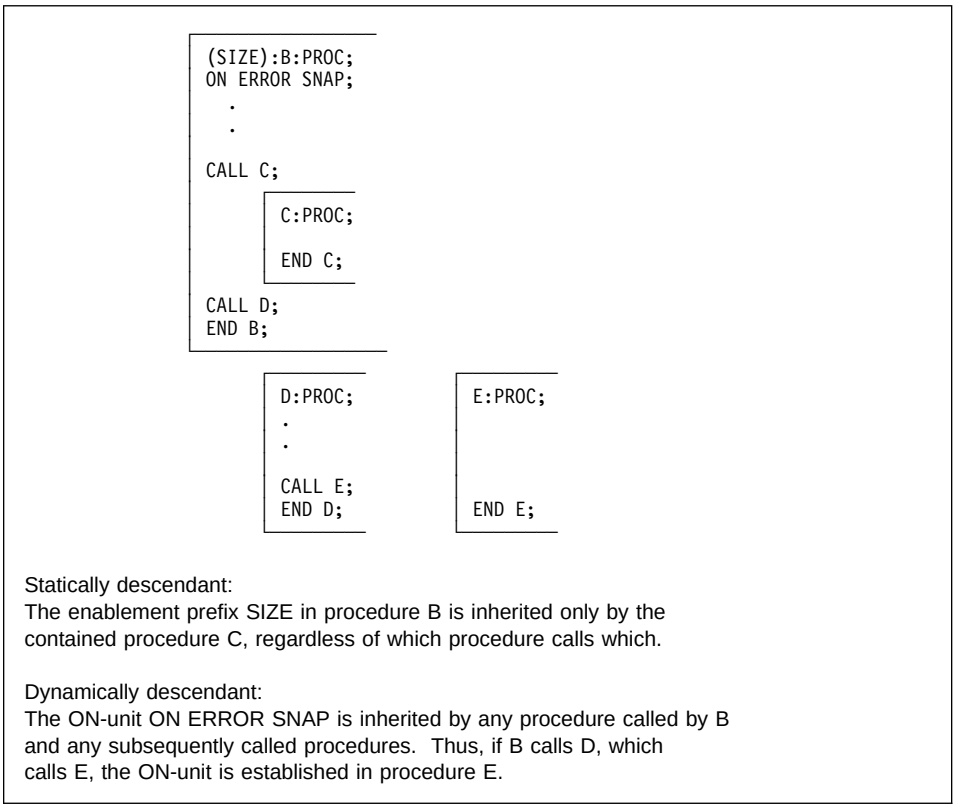


Figure 7. Static and dynamic descendant procedures

Normal return

A normal return is a return from a called block after reaching the END or RETURN statement, rather than reaching a GOTO statement out of a block. In an error-handling context, normal return is taken to mean normal return from the ON-unit. The action taken after normal return from an ON-unit is specified in the *PL/I Language Reference*.

Standard system action
Standard system action refers to the default PL/I-defined action taken when a condition occurs for which there is no established ON-unit.

Error handling concepts

You should be familiar with the following error handling concepts when you attempt to debug your PL/I programs. For details on condition handling, see the *PL/I Language Reference*.

System facilities

The operating system offers error-handling facilities. Various situations can cause a machine interrupt, resulting in an entry to the system supervisor. The PL/I control program can use specified routines to define the action that is taken after any of these interrupts. Alternatively, the PL/I control program passes control to ON-units specified by the PL/I programmer.

Language facilities

The PL/I language and its execution environment extend the error-handling facilities offered by the operating system. Numerous situations can cause interrupts for PL/I, and some situations (such as ENDFILE) can be used to control normal program flow rather than to handle errors. ON-units allow you to obtain control after most interrupts.

If you do not write ON-units to obtain control after interrupts, you can:

- Accept standard system action
- Choose whether certain conditions cause interrupts or not by enabling or disabling those conditions. If the condition is disabled, neither ON-unit nor standard system action is taken when the condition occurs.

The majority of PL/I conditions occur because of errors in program logic or the data supplied. Some, however, are not connected with errors. These are conditions such as ENDFILE, which are difficult to anticipate because they can occur at any time during program execution.

PL/I has both system messages and snap messages:

System messages
If an ON-unit contains both SNAP and SYSTEM, the resulting PL/I message is essentially the PL/I SYSTEM message followed by any (or a combination) of the following three lines:

```
From offset xxx in a BEGIN block

From offset xxx in procedure xxx

From offset xxx in a condition_name ON-unit
```

These messages are repeated as often as necessary to trace back to the main procedure.

SNAP messages

If an ON-unit contains only SNAP, the resulting PL/I message begins:

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

The messages then continue as for SNAP SYSTEM messages.

Determining statement numbers from offsets: If you want to translate offset numbers into statement numbers, use the following steps:

- Use the OFFSET compile-time option during compilation
- Open the resulting object (`.cod`) listing file
- Search for and locate the offset in the first column and find the statement number from the last source statement included in the listing.

Built-ins for condition handling: PL/I also provides condition-handling built-in functions and pseudovariables. These allow you to inspect various fields associated with the interrupt and, in certain cases, to correct the contents of fields causing the error.

These built-in functions include:

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

For detailed information on these condition-handling built-in functions and pseudovariables, consult the *PL/I Language Reference*.

ON-units for qualified and unqualified conditions

There can only be one established ON-unit for an unqualified condition at any given point in a program, but there can be more than one established ON-unit for qualified conditions. For example, in handling the ENDFILE condition as qualified for different files, you can have an ON-unit established to uniquely handle the occurrence of ENDFILE for any one of the files.

Conditions used for testing and debugging

The following conditions are useful in testing and debugging your programs:

- SUBSCRIPTRANGE
- STRINGSIZE
- STRINGRANGE

Running your program with these conditions decreases performance, but ON-units for these conditions can serve as powerful tools for finding out the sources of errors in your program. You can enable any of these conditions by writing an ON-unit for them. Then, if the condition is raised, your ON-unit can define an action that tells you the cause of the error.

For example, if your program raises FIXEDOVERFLOW, it is useful to issue PUT DATA to discover the values of your data that led to the condition being raised.

In addition, the PREFIX option is useful because you can enable conditions without having to edit your program.

Common programming errors

A failure in running a PL/I program can be caused by:

- Logical errors in a source program
- Invalid use of PL/I (for example, uninitialized variables)
- Calling uninitialized entry variables
- Loops and other unforeseen errors
- Unexpected input/output data
- Unexpected program termination
- Other unexpected program results
- System failure
- Poor performance

Logical errors in your source programs

Logical errors in a source program are often difficult to detect and sometimes can make it appear as though there are compiler or library failures.

Some common errors in source programs are:

- Failure to convert correctly from arithmetic data
- Incorrect arithmetic and string-manipulation operations
- Failure to match data lists with their format lists

Invalid use of PL/I

A misunderstanding of the language can result in an apparent program failure. For example, any of the following programming errors can cause a program to fail:

- Using uninitialized variables
- Using controlled variables that have not been allocated
- Reading records into incorrect structures
- Misusing array subscripts
- Misusing pointer variables
- Incorrect conversion
- Incorrect arithmetic operations
- Incorrect string-manipulation operations
- Freeing or using storage that was never allocated or already free

Calling uninitialized entry variables

If you call an entry variable that is uninitialized:

- OS/2 and Windows NT raise a protection exception almost immediately.

- Windows 95, however, does not raise an immediate protection exception and allows you to execute instructions in low memory which can cause unpredictable program behavior.

Loops and other unforeseen errors

If an error is detected during execution of a PL/I program, and no ON-unit is provided in the program to terminate execution or attempt recovery, the job terminates abnormally. However, you can record the status of your program at the point where the error occurred by using an ERROR ON-unit that contains the statements:

```
on error
 begin;
  on error system;
  call plidump ('TFBS','This is a dump');
 end;
```

The statement ON ERROR SYSTEM; contained in the ON-unit ensures that further errors caused by attempting to transmit uninitialized variables do not result in an endless loop.

If you want to take action based on the specific type of condition being handled, use the ONCONDID function (for more information on this function, see the *PL/I Language Reference*):

```
on anycondition
 begin;
  on anycondition system;
  select( oncondid() );
    when( condid_ofl )
     ⋮
    when( condid_ufl )
     ⋮
    when( condid_zdiv )
     ⋮
    otherwise
      resignal;
  end;
 end;
```

Tips for dealing with loops

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

```
on Error begin;
  on Error System;
   ⋮
end;
```

If your program is caught in an endless loop, your primary concern is to be able to get out of the loop without shutting down your machine. The following solution is recommended for handling endless loops:

Common programming errors

- OS/2 Include an ON-unit in your program that calls PLIDUMP if the ATTENTION condition is raised, for example: `on attention snap;`. For information on debugging with dumps, see “Using dumps for debugging” on page 188.
- When the loop is entered, hit **Ctrl-Break**. This raises the ATTENTION condition and produces the dump by means of the ON-unit you have added. If no ATTENTION ON-unit is present, the program ends.
- Examine the dump. Use the trace information to see where the loop originated. Correct the program so that it does not produce the loop.
- WIN When the loop is entered, hit **Ctrl-Break** to end your program. No ATTENTION ON-unit is driven in this environment.

Unexpected input/output data

A program should contain checks to ensure that any incorrect input and output data is detected before it can cause the program to fail.

Use the COPY option of the GET and PUT statements if you want to check values obtained by stream-oriented input and output. The values are listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

Use the VALID built-in function to check the validity of PICTURE and FIXED DECIMAL identifiers.

For additional information on features that can lead to unexpected I/O, see Chapter 2, “Porting applications between platforms” on page 7. Many of the features that can lead to portability problems (such as differences in ASCII and EBCDIC collating sequences) can also lead to unexpected I/O for your PL/I programs.

Unexpected program termination

If your program terminates abnormally without an accompanying run-time diagnostic message, the error that caused the failure probably also prevented the message from being displayed. Possible causes of this type of behavior are:

- Trying to run modules that were not compiled by this version of the compiler.
- Incorrect export DD= statements.

- Overwriting storage areas that contain executable instructions, particularly the PL/I communications area. Any of the following could cause overwriting of storage areas:
  - Assigning a value to a nonexistent array element. For example:

    ```
    dcl array(10);
       ⋮
    do I = 1 to 100;
    array(I) = value;
    ```

    You can detect this type of error in a compile module by enabling the SUBSCRIPTRANGE condition. Each attempt to access an element outside the declared range of subscript values should raise the SUBSCRIPTRANGE condition. If there is no ON-unit for this condition, a diagnostic message prints and the ERROR condition is raised.

    Though this method is costly in terms of execution time and storage space, it is a valuable program testing aid. For more information on error handling, see “Using error and condition handling for debugging” on page 192.
  - Using an incorrect locator value for a locator (pointer or offset) variable. This type of error is possible if a locator value is obtained using a record-oriented transmission.

    Make sure that locator values created in one program, transmitted to a data set, and subsequently retrieved for use in another program, are valid for use in the second program.
  - Attempting to free a non-BASED variable. This can happen when you free a BASED variable after its qualifying pointer value has been changed. For example:

    ```
    dcl a static,b based (p);
    allocate b;
    p = addr(a);
    free b;
    ```
  - Using an incorrect value for a label, entry, or file variable. Label, entry, and file values that are transmitted and subsequently retrieved are subject to the same kind of errors as those described previously for locator values.
  - Using the SUBSTR pseudovariable to assign a string to a location beyond the end of the target string. For example:

    ```
    dcl x char(3);
    i = 3
    substr(x,2,i) = 'ABC';
    ```

    To detect this type of error in a compiled module, use the STRINGRANGE condition (for more information, see “Conditions used for testing and debugging” on page 195).

Common programming errors

Other unexpected program results

WIN Due to a difference in the way OS/2 and Windows respond to floating-point conditions, you might experience altered program flow. One consequence of altered program flow is conditions that do not get raised because they have become disabled.

For example, although using the NOIMPRECISE compile-time option does provide better floating-point error detection than IMPRECISE, the Windows operating system does not always detect floating-point exceptions immediately. If you have a statement in your program that is likely to raise a floating-point exception, you can avoid this detection problem by enclosing the statement, by itself, in a BEGIN block.

Compiler or library subroutine failure

If you are convinced that the failure is caused by a compiler failure or a library subroutine failure, you should contact IBM.

Meanwhile, you can attempt to find an alternative way to perform the operation that is causing the trouble. A bypass is often possible because the PL/I language frequently provides an alternative method of performing a given operation.

System failure

System failures include machine malfunctions and operating system errors. System messages identify these failures to the operator.

Poor performance

While not necessarily caused by bugs, poor performance is associated with excessive run-time and memory requirements. One thing to keep in mind is that many debugging techniques (such as enabling SUBSCRIPTRANGE) tend to decrease performance.

One feature that can increase performance is the OPTIMIZE compile-time option (see “OPTIMIZE” on page 66). For additional information on improving program performance, see Chapter 19, “Improving performance” on page 359.

Part 4. Input and output

Chapter 11. Using data sets and files

Types of data sets

Your PL/I programs can process and transmit units of information called *records*. A collection of records is called a data set, but for PL/I workstation products, a data set can be either a file or a device. Data sets are logical collections of information external to PL/I programs; they can be created, accessed, or modified by programs written in PL/I.

Your PL/I program recognizes and processes information in a data set by associating it with a symbolic representation of the data set called a PL/I file. This PL/I file represents the environment independent characteristics of a set of input and output operations.

In order to minimize confusion, this book uses the term *PL/I file* to refer to the file declared and used in a PL/I program. The terms data set and workstation file (or workstation device) are used to refer to the collection of data on an external I/O device. In some cases the data sets have no name; they are known to the system by the device on which they exist.

Types of data sets

PL/I defines two types of data sets—native data sets and workstation VSAM data sets.

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

Platform distinctions

This chapter refers to the access methods available on PL/I workstation products; however, all methods are not available on all platforms. As you refer to information in this chapter, use the following guideline:

- DDM—supported on AIX and OS/2 only
- ISAM—supported on AIX, OS/2, and Windows
- BTRIEVE—supported on OS/2 and Windows only
- REMOTE—supported on OS/2 and Windows to access mainframe data files

To convert mainframe VSAM files to the corresponding DDM, ISAM, or BTRIEVE files, follow the procedure documented in the prolog for the LODVSAM utility (not yet supported on AIX). Make sure you specify the appropriate access method AMTHD(DDM|ISAM|BTRIEVE).

To convert DDM, ISAM, or BTRIEVE files to corresponding mainframe VSAM files, follow the procedure documented in the prolog for the RELOAD utility (not yet supported on AIX). These utilities should be in the samples directory for VisualAge PL/I.

Types of data sets

Data sets that reside on the mainframe can be accessed remotely by your PL/I program using the Distributed FileManager product that comes with SMARTdata Utilities (SdU), one of the VisualAge PL/I components. You can find information about using SdU in the online books for that product. The online books for SdU are installed only if you select that component.

OS/2 For OS/2, refer to *VSAM in a Distributed Environment*, especially the sections on the Distributed FileManager for OS/2. This information includes an introduction and administrative activities.

WIN For Windows, refer to the *Distributed FileManager User's Guide*.

There are several types of native data sets:

- Conventional text files
- Character devices
- Fixed-length data sets

Both record and stream I/O can be used to access these types of data sets, which can be accessed only in a sequential manner.

Additional types of PL/I-defined data sets include:

- Varying-length
- Regional
- Workstation VSAM data sets

Only record I/O can be used to access regional data sets. Access can be either sequential or direct.

Native data sets

A native data set in PL/I terms defines conventional text files and devices associated with the platform you are using.

Conventional text files and devices

A conventional text file has logical records delimited by the CR - LF (carriage return and line feed) character sequence. Most text editor programs create, and allow you to alter, conventional text files. Your PL/I programs can create conventional text files, or they can access text files that were created by other programs.

Devices for workstation products are the keyboard, screen, and printer. The names you use to refer to them in PL/I are:

OS/2 KBD$: (or KBD$)	Keyboard
OS/2 SCREEN$: (or SCREEN$)	Screen
OS/2 CON: (or CON)	Console (the keyboard for input, the screen for output)
OS/2 PRN: (or PRN)	Printer (defaults to LPT1)
OS/2 LPT1: (or LPT1)	Parallel printer port
OS/2 LPT2: (or LPT2)	Parallel printer port
OS/2 LPT3: (or LPT3)	Parallel printer port
NUL: (or NUL)	Null output device (for discarding output)

Types of data sets

STDIN:	Standard input file (defaults to CON)
STDOUT:	Standard output file (defaults to CON)
STDERR:	Standard error message file (defaults to CON)

Note: STDIN:, STDOUT:, and STDERR: can be redirected, whereas the other device names cannot. The mouse is only accessed through PM; therefore, it is not listed above.

Fixed-length data sets

PL/I also allows you to treat a file as a set of fixed-length records. Your PL/I programs can create fixed-length data sets, or access existing files as fixed-length data sets. The data access does not treat Carriage Return(CR) or Line Feed (LF) as characters with special meaning. In particular, the CR - LF sequence does not delimit records, although these characters can be contained in the data set. It is the length you specify that determines what PL/I considers to be a record within the data set. This type of data set has the restriction that the total number of characters in the data set must be evenly divisible by the length you specify.

Fixed-length data sets can be accessed only in a sequential manner.

Additional data sets

Other types of data sets include varying-length, regional, and workstation VSAM data sets.

Varying-length data sets

Your PL/I program can also create and access data sets where each record has a two-byte prefix that specifies the number of bytes in the rest of the record. Unlike files with records delimited by CR - LF, these varying-length files can have records that possibly contain arbitrary bit patterns.

Regional data sets

A description of regional data sets and how you can use them is presented in Chapter 13, “Defining and using regional data sets” on page 258.

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

Workstation VSAM data sets

The PL/I workstation products support VSAM file organization. There are three types of VSAM data sets on the workstation:

- Consecutive, similar to a VSAM entry-sequenced data set (ESDS)
- Relative, similar to a VSAM relative record data set (RRDS)
- Indexed, similar to a VSAM key-sequenced data set (KSDS)

The PL/I workstation products currently support the following methods for accessing VSAM data sets:

- DDM (AIX and OS/2 only)
- ISAM (AIX, OS/2, and Windows)

- BTRIEVE (OS/2 and Windows only)
- REMOTE to access mainframe data files on OS/2 and Windows

Throughout the remainder of the chapter, the discussion generally applies to DDM, ISAM, and BTRIEVE unless otherwise noted.

DDM access method (AIX and OS/2 only)
DDM data sets are record-oriented files as defined by the Distributed Data Management Architecture. Workstation VSAM data sets that use the DDM access method can exist on local systems. You can compile and run most existing mainframe programs that reference mainframe VSAM data sets.

A DDM keyed data set is represented by two files—one called the *base*, and the other called the *prime index*. The records of the data set are kept in the base; the prime index contains information about the primary keys of the data set.

When you create a DDM keyed data set, you specify the name of the base; DDM generates a name for the prime index, which it derives from the name of the base.

When you use DDM data sets, you do not need to be concerned about record length, except that your records cannot exceed the maximum specified length.

You can compile and run most existing mainframe programs that reference mainframe VSAM data sets by either:

- Creating the appropriate workstation VSAM data set on your PC before running the program
- Remotely accessing the data files on the mainframe

ISAM access method (AIX, OS/2, and Windows)
Unless otherwise specified, the term *ISAM* in this chapter refers to the ISAM local access method and not mainframe ISAM. ISAM data sets are stored in one file and can exist on local file systems only.

BTRIEVE access method (OS/2 and Windows only)
The BTRIEVE access method is provided to allow you to use PL/I input and output statements to access files created under CICS. There is currently no PL/I support for BTRIEVE segmented and multiple keys.

BTRIEVE data sets are stored in one file and can exist on local file systems only.

REMOTE access method on OS/2 and Windows
The REMOTE access method is provided to allow you to remotely access data files on the mainframe.

Detailed information on workstation VSAM is found in Chapter 14, “Defining and using workstation VSAM data sets” on page 270.

Establishing data set characteristics

When you declare or open a file in your program, you are describing to PL/I the characteristics of the file. You can also use a DD:ddname environment variable or an expression in the TITLE option of the OPEN statement to describe to PL/I the

Establishing data set characteristics

characteristics of the data in data sets or in PL/I files associated with them. See “Associating a PL/I file with a data set” on page 225 for more information.

You do not always need to describe your data both within the program and outside it; often one description serves for both data sets and their associated PL/I files. There are, in fact, advantages to describing your data's characteristics in only one place. These are described later in this chapter and in following chapters.

To effectively describe your program data and the data sets you are using, you need to understand something about how PL/I moves and stores data.

Records

A record is the unit of data transmitted to and from a program. You can specify the length of records in the RECSIZE option for any of the following:

DD information
PL/I ENVIRONMENT attribute
TITLE option of the OPEN statement

Except for certain stream files, where defaults are applied, you must specify the RECSIZE option when your PL/I program creates a data set. For more information about stream files, see Chapter 12, “Defining and using consecutive data sets” on page 233.

You must also specify the RECSIZE option when your program accesses a data set that was not created by PL/I.

Please note that an editor might alter a data set implicitly. You should use special caution if you examine a non CR - LF file using an editor, because most editors automatically insert CR - LF or similar character sequences.

Record formats

The records in a data set can have one of the following formats:

Undefined-length
Fixed-length
Varying-length

For a native file, you specify either undefined-length or fixed-length record format in the TYPE option of the DD information. You do not need to specify a record format for workstation VSAM data sets; they implicitly consist of varying-length records.

Data set organizations

The options of the PL/I ENVIRONMENT attribute that specify data set organization are:

CONSECUTIVE
ORGANIZATION(CONSECUTIVE)
ORGANIZATION(INDEXED)
ORGANIZATION(RELATIVE)
REGIONAL(1)

VSAM

Each is described in “Specifying characteristics using the PL/I ENVIRONMENT attribute.”

If you do not specify the data set organization option in the ENVIRONMENT attribute, it defaults to CONSECUTIVE.

Specifying characteristics using the PL/I ENVIRONMENT attribute

The ENVIRONMENT attribute of the DECLARE statement allows you to specify certain data set characteristics within your programs. These characteristics are not part of the PL/I language; hence, using them in a file declaration might make your program non-portable to other PL/I implementations.

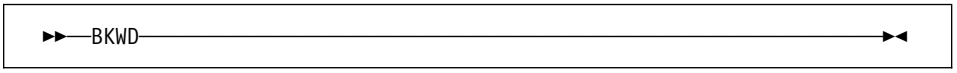
Here is an example of how to specify environment options for a file in your program:

```
declare Invoices file environment(regional(1), recsize(64));
```

The options you can specify in the ENVIRONMENT attribute are defined in the following sections.

BKWD

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



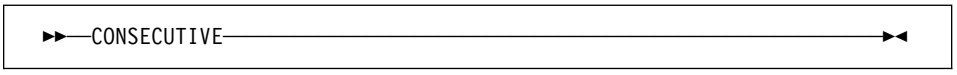
Sequential reads (that is, reads without the KEY option) retrieve the previous record in sequence. For indexed data sets, the previous record is the record with the next lower key.

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

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

CONSECUTIVE

The CONSECUTIVE option defines a file with consecutive data set organization. In a data set with CONSECUTIVE organization, records are placed in physical sequence. Given one record, the location of the next record is determined by its physical position in the data set.



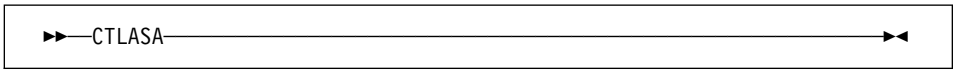
Establishing data set characteristics

You use the CONSECUTIVE option to access native data sets using either stream-oriented or record-oriented data transmission. You also use it for input files declared with the SEQUENTIAL attribute and associated with a workstation VSAM data set. In this case, records in a workstation VSAM keyed data set are presented in key sequence.

CONSECUTIVE is the default data set organization.

CTLASA

The CTLASA option specifies that the first character of a record is to be interpreted as an American National Standard (ANS) print control character. The option applies only to RECORD OUTPUT files associated with consecutive data sets.



The ANS print control characters, listed in Table 15 on page 234, cause the specified action to occur before the associated record is printed.

For information about how you use the CTLASA option, see “Printer-destined files” on page 234.

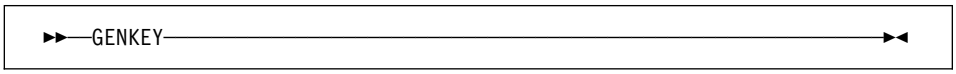
The IBM Proprinter control characters require up to 3 bytes more than the single byte required by an ANS printer control character. However, *do not* adjust your logical record length specification (see the RECSIZE environment option) because PL/I automatically adds 3 to the logical record length when you specify CTLASA.

You can modify the effect of CTLASA so that the first character of records is left untranslated to IBM Proprinter control characters. See the ASA environment option under “ASA” on page 217.

Do not specify the SCALARVARYING environment option for printer-destined output operations, as PL/I does not know how to interpret the first data byte of records.

GENKEY

The GENKEY (generic key) option applies only to workstation VSAM indexed data sets. It enables you to classify keys recorded in the data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key class.



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

recorded keys can be considered to be unique members of the classes “ABCD,” “ABCE,” and “ABDF,” respectively.

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an INDEXED data set from the first nondummy record that has a key in a particular class. You identify the class by including its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although you can retrieve the first record having a key in a particular class by using a READ with the KEY option, you cannot obtain the actual key unless the records have embedded keys, since the KEYTO option cannot be used in the same statement as the KEY option.

In the following example, a key length of more than three bytes is assumed:

```
dcl ind file record sequential keyed
  update env (indexed genkey);
             ⋮
    read file (ind) into (infield)
                    key ('ABC');
             ⋮
next: read file (ind) into (infield);
             ⋮
  go to next;
```

The first READ statement causes the first nondummy record in the data set with a key beginning 'ABC' to be read into INFIELD. Each time the second READ statement is executed, the nondummy record with the next higher key is retrieved. Repeated execution of the second READ statement could result in reading records from higher key classes, since no indication is given when the end of a key class is reached. It is your responsibility to check each key if you do not wish to read beyond the key class. Any subsequent execution of the first READ statement would reposition the file to the first record of the key class 'ABC'.

If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

The presence or absence of the GENKEY option affects the execution of a READ statement which supplies a source key that is shorter than the key length specified in the KEYLENGTH subparameter. The KEYLENGTH subparameter is found in the DD statement that defines the indexed data set. If you specify the GENKEY option, it causes the source key to be interpreted as a generic key, and the data set is positioned to the first nondummy record in the data set whose key begins with the source key.

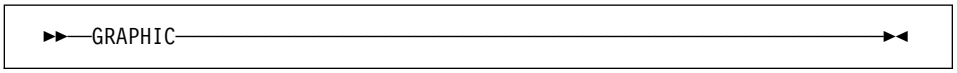
If you do not specify the GENKEY option, a READ statement's short source key is padded on the right with blanks to the specified key length, and the data set is

positioned to the record that has this padded key (if such a record exists). For a WRITE statement, a short source key is always padded with blanks.

Use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered the only member of its class).

GRAPHIC

You must specify the GRAPHIC option if you use DBCS variables or DBCS constants in GET and PUT statements for list-directed and data-directed I/O. You can also specify the GRAPHIC option for edit-directed I/O.

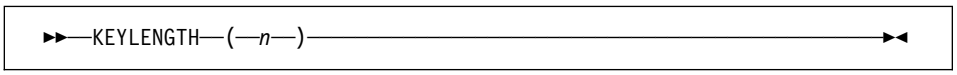


PL/I raises the ERROR condition for list-directed and data-directed I/O if you have graphics in input or output data and you do not specify the GRAPHIC option.

For information on the graphic data type, and on the G-format item for edit-directed I/O, see the *PL/I Language Reference*.

KEYLENGTH

The KEYLENGTH option specifies the length, *n*, of the recorded key for a KEYED file. You can specify KEYLENGTH only for INDEXED files (see ORGANIZATION later in this section).



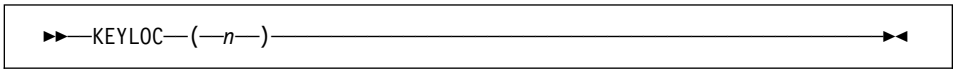
If you include the KEYLENGTH option in a file declaration, and the associated data set already exists, the value is used for checking purposes. If the key length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

ISAM and BTRIEVE

Keys are kept in the index pages of an ISAM or BTRIEVE file. The length of the key needs to be defined to PL/I when the file is created.

KEYLOC

The KEYLOC option specifies the starting position, *n*, of the embedded key in records of a KEYED file. You can specify KEYLOC only for INDEXED files (see ORGANIZATION later in this section).



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

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

That is, the key cannot be larger than the record and must be contained completely within the record.

This means that if you specify the SCALARVARYING option, the embedded key must not overlap the first two bytes of the record; hence, the value you specify for KEYLOC must be greater than 2.

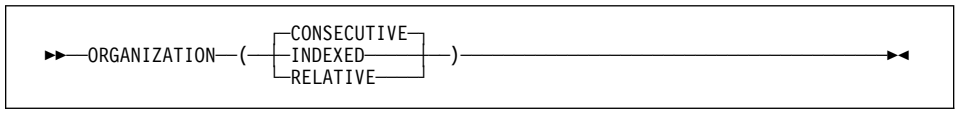
If you do not specify KEYLOC when creating an indexed data set, the key is assumed to start with the first byte of the record.

If you include the KEYLOC option in a file declaration, and the associated data set already exists, the value is used for checking purposes. If the key position you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

ISAM and BTRIEVE
Keys are kept in the index pages of an ISAM or BTRIEVE file. The location of the key needs to be defined to PL/I when the file is created.

ORGANIZATION

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



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

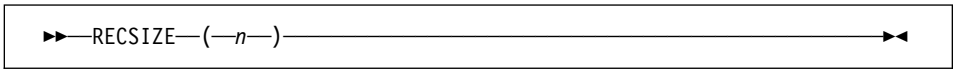
INDEXED
Specifies that the file is associated with an indexed data set. INDEXED specifies that the data set contains records arranged in a logical sequence, according to keys embedded in each record. Logical records are arranged in the data set in ascending key sequence according to the ASCII collating sequence. An indexed file is a workstation VSAM keyed data set.

RELATIVE
Specifies that the file is associated with a relative data set. RELATIVE specifies that the data set contains records that do not have recorded keys. A relative file is a workstation VSAM direct data set. Relative keys range from 1 to nnnn.

Establishing data set characteristics

RECSIZE

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



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

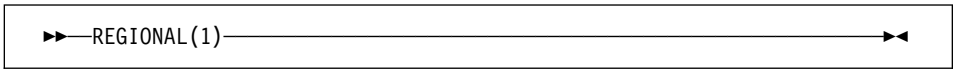
If you include the RECSIZE option in a file declaration, and the file is associated with a workstation VSAM data set that already exists, the value is used for checking purposes. If the record length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

Specify the RECSIZE option when you access data sets created by non-PL/I programs such as text editors.

ISAM and BTRIEVE
You must specify RECSIZE when using the BTRIEVE or ISAM access method.

REGIONAL(1)

The REGIONAL(1) option defines a file with the regional organization.

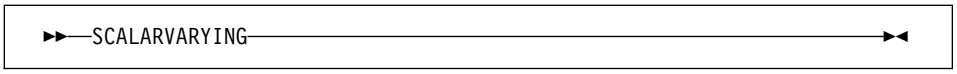


A data set with regional organization contains fixed-length records that do not have recorded keys. Each region in the data set contains only one record; therefore, each region number corresponds to a relative record within the data set (that is, region numbers start with 0 at the beginning of the data set).

For information about how you use regional data sets, see Chapter 13, “Defining and using regional data sets” on page 258.

SCALARVARYING

The SCALARVARYING option is used in the input and output of VARYING strings.



When storage is allocated for a VARYING string, the compiler includes a 2-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if you specify SCALARVARYING for the file.

When you use locate mode statements (LOCATE and READ SET) to create and read a data set with element VARYING strings, you must specify SCALARVARYING to indicate that a length prefix is present, since the pointer that locates the buffer is always assumed to point to the start of the length prefix.

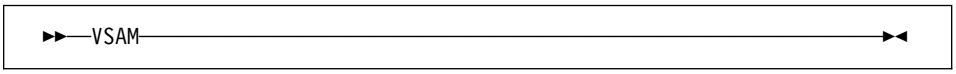
When you specify this option and element VARYING strings are transmitted, you must allow two bytes in the record length to include the length prefix.

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

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

VSAM

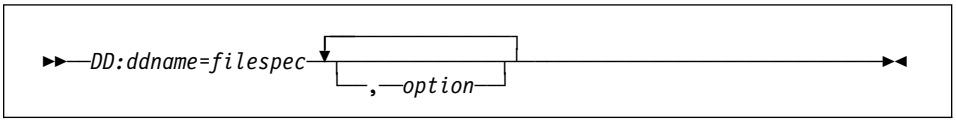
The VSAM option is provided for compatibility with OS PL/I.



Specifying characteristics using DD:ddname environment variables

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

The syntax of the DD:ddname environment variable is:



Blanks are acceptable within the syntax. In addition, the syntax of the statement is not checked at the time the command is entered. It is verified when the data set is opened. If the syntax is wrong, UNDEFINEDFILE is raised with the oncode 96.

DD:ddname

Specifies the name of the environment variable. The ddname can be either the name of a file constant or an alternate ddname that you specify in the TITLE option of your OPEN statement. The TITLE option is described in “Using the TITLE option of the OPEN statement” on page 225.

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.

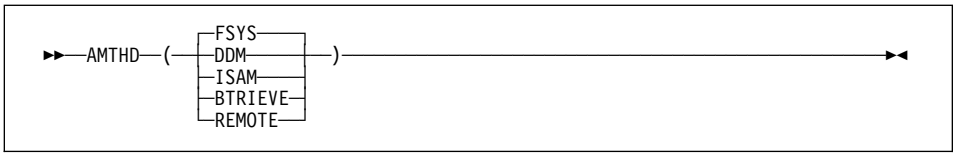
filespec
Specifies a file or the name of a device to be associated with the PL/I file. See “Conventional text files and devices” on page 205 for the names you use to specify the appropriate device.

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 “AMTHD” and ending with “TYPE” on page 223.

AMTHD

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



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

DDM(AIX and OS/2 only)
Specifies that the DDM access method is to be used to access a DDM file.

ISAM (AIX, OS/2, and Windows)
Specifies that the ISAM access method is to be used to access an ISAM file.

BTRIEVE (OS/2 and Windows)
Specifies that the BTRIEVE access method is to be used to access a BTRIEVE file.

REMOTE (OS/2 and Windows)
Specifies that the file resides on a remote DDM target system (such as MVS).

OS/2 For OS/2, the name of the file needs to be qualified by the drive letter to which the target system is mapped.

WIN For Windows, the name of the file needs to be qualified by the LU alias or the fully-qualified SNA network name.

FSYS is used by default if you do not specify the AMTHD option and if you do not apply one of the following ENVIRONMENT options:

ORGANIZATION(INDEXED)
ORGANIZATION(RELATIVE)
VSAM

If you specify any of the above options, AMTHD(ISAM)is the default on OS/2 and Windows while AMTHD(DDM) is the default on AIX.

Have you installed the correct file system?

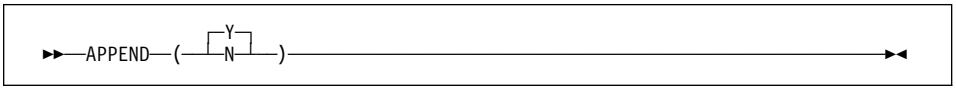
Before you apply any of the previously listed options, make sure you have installed the appropriate file system:

- DDM—supported on AIX and OS/2 only
- ISAM—supported on AIX, OS/2, and Windows
- BTRIEVE—supported on OS/2 and Windows only
- REMOTE—support for accessing mainframe data files from OS/2 and Windows.

Otherwise, the UNDEFINEDFILE condition is raised.

APPEND

The APPEND option specifies whether an existing data set is to be extended or re-created.



Y Specifies that new records are to be added to the end of a sequential data set, or inserted in a relative or indexed data set. This is the default.

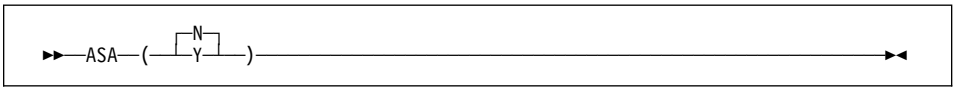
N Specifies that, if the file exists, it is to be re-created.

The APPEND option applies only to OUTPUT files. APPEND is ignored if:

- The file does not exist
- The file does not have the OUTPUT attribute
- The organization is REGIONAL(1)

ASA

The ASA option applies to printer-destined files. This option specifies when the ANS control character in each record is to be interpreted.



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

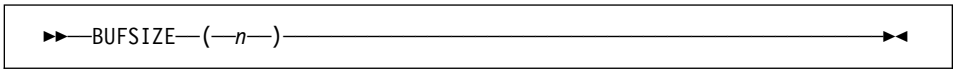
Y Specifies that the ANS print control characters are not to be translated; instead they are to be left as is for subsequent translation by a process you determine.

If the file is not a printer-destined file, the option is ignored. Printer-destined files are described in “Printer-destined files” on page 234.

Establishing data set characteristics

BUFSIZE

The BUFSIZE option specifies the number of bytes for a buffer.



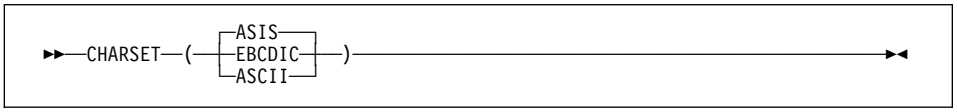
RECORD output is buffered by default and has a default value for BUFSIZE of 64k. STREAM output is buffered, but not by default, and has a default value for BUFSIZE of zero.

If the value of zero is given to BUFSIZE, the number of bytes for buffering is equal to the value specified in the RECSIZE or LRECL option.

The BUFSIZE option is valid only for a consecutive binary file. If the file is used for terminal input, you should assign the value of zero to BUFSIZE for increased efficiency.

CHARSET for record I/O

This version of the CHARSET option applies only to consecutive files using record I/O. It gives the user the capability of using EBCDIC data files as input files, and specifying the character set of output files.

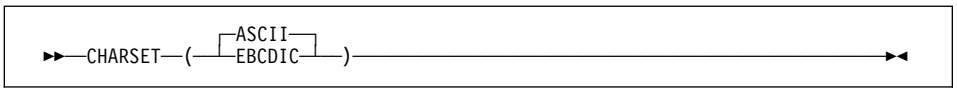


Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file have (output).

CHARSET(ASIS) is the default.

CHARSET for stream I/O

This version of the CHARSET option applies for stream input and output files. It gives the user the capability of using EBCDIC data files as input files, and specifying the character set of output files. If you attempt to specify ASIS when using stream I/O, no error is issued and character sets are treated as ASCII.

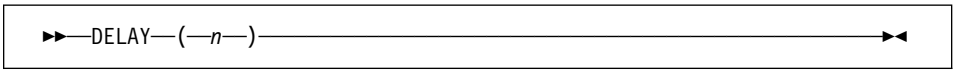


Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file to have (output).

CHARSET(ASCII) is the default.

DELAY

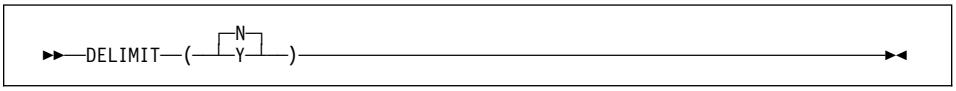
The DELAY option specifies the number of milliseconds to delay before retrying an operation that fails when a file or record lock cannot be obtained by the system.



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

DELIMIT

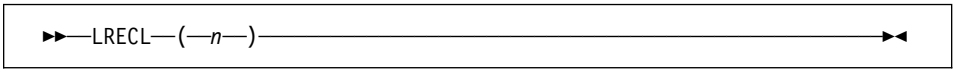
The DELIMIT option specifies whether the input file contains field delimiters or not. A field delimiter is a blank or a user-defined character that separates the fields in a record. This is applicable for sort input files only.



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

LRECL

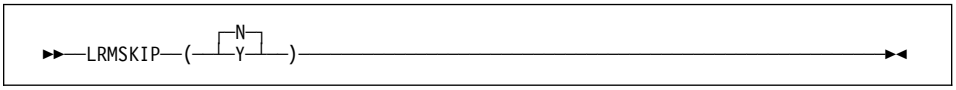
The LRECL option is the same as the RECSIZE option.



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

LRMSKIP

The LRMSKIP option allows output to commence on the `nth` (`n` refers to the value specified with the SKIP option of the PUT or GET statement) line of the first page for the first SKIP format item to be executed after a file is opened.

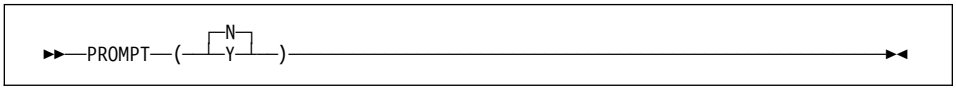


If `n` is zero or 1, output commences on the first line of the first page.

PROMPT

The PROMPT option specifies whether or not colons should be visible as prompts for stream input from the terminal.

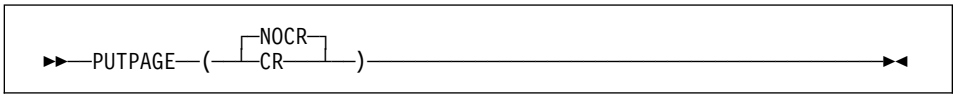
Establishing data set characteristics



PROMPT(N) is the default.

PUTPAGE

The PUTPAGE option specifies whether or not the form feed character should be followed by a carriage return character. This option only applies to printer-destined files. Printer-destined files are stream output files declared with the PRINT attribute, or record output files declared with the CTLASA environment option.

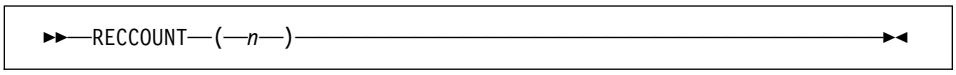


NOCR
Indicates that the form feed character ('0C'x) is not followed by a carriage return character ('0D'x). This is the default.

CR Indicates that the carriage return character is appended to the form feed character. This option should be specified if output is sent to non-IBM printers.

RECCOUNT

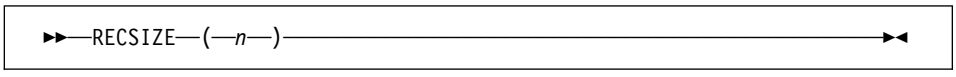
The RECCOUNT option specifies the maximum number of records that can be loaded into a relative or regional data set that is created during the PL/I file opening process.



The RECCOUNT option is ignored if PL/I does not create, or re-create, the data set. If the RECCOUNT option applies and is omitted, the default is 50 for regional and relative files.

RECSIZE

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

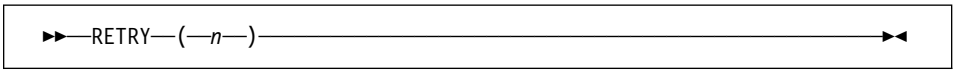


For regional and fixed-length data sets, RECSIZE specifies the length of each record in the data set; for all other data set types, RECSIZE specifies the maximum length records may have.

The default for *n* is 512.

RETRY

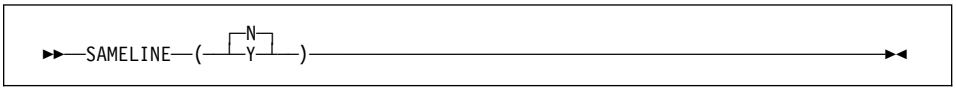
The RETRY option specifies the number of times an operation should be retried when a file or record lock cannot be obtained by the system.



The default value for RETRY is 10. This option is applicable only to DDM files.

SAMELINE

The SAMELINE option specifies whether the system prompt occurs on the same line as the statement that prompts for input.



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

Example 1

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

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

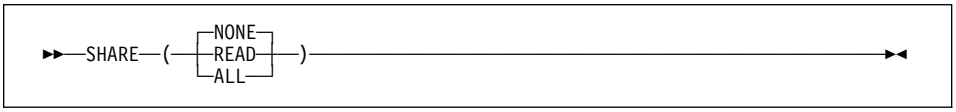
Example 2

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

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

SHARE

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



NONE
Specifies that the file is not to be shared with other processes. This is the default.

READ
Specifies that other processes can read the file.

ALL
Specifies that other processes can read or write the file. Data integrity is the user's responsibility, and PL/I provides no assistance in maintaining it.

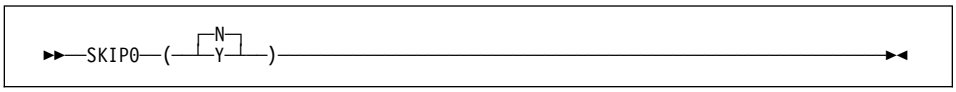
This option is valid only with DDM files.

To enable record-level locking, specify SHARE(ALL) and declare the file as an update file. This is recommended when running CICS applications.

The UNDEFINEDFILE condition is raised if the requested or default level of file sharing cannot be obtained.

SKIP0

The SKIP0 option specifies where the line cursor moves when SKIP(0) statement is coded in the source program. SKIP0 applies to terminal files that are not linked as PM applications.



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

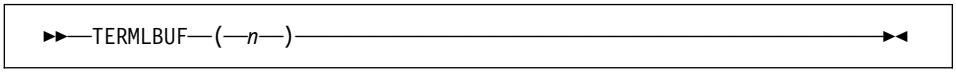
SKIP0(Y)
Specifies that the cursor to be moved to the beginning of the current line.

The following example shows how you could make the output to the terminal skip zero lines so that the cursor moves to the beginning of the current output line:

```
set dd:sysprint=stdout:,SKIP0(Y)
set dd:sysprint=con,SKIP0(Y)
```

TERMLBUF

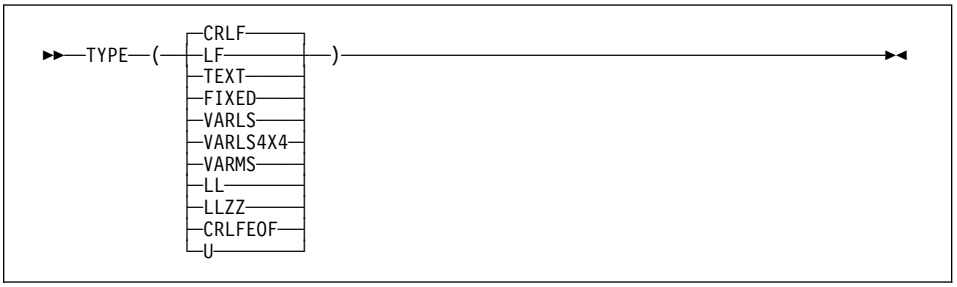
The TERMLBUF option specifies the maximum number of lines in the window of a PL/I Presentation Manager (PM) terminal.



If the file is not associated with a PM terminal, the option is ignored. The default is 512 lines.

TYPE

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



CRLF
Specifies that records are delimited by the CR - LF character combination. ('CR' and 'LF' represent the ASCII values of carriage return and line feed, '0D'x and '0A'x, respectively. See restrictions on 15) 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.

This is the default for ISAM and BTRIEVE.

LF Specifies that records are delimited by the LF character combination. ('LF' represents the ASCII values of feed or '0A'x. See restrictions on 15) For an output file, PL/I places the characters at the end of each record; for an input file, PL/I discards the characters. For both input and output, the characters are not counted in consideration for RECSIZE.

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

TEXT
Equivalent to CRLF.

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

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

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

TYPE(VARLS) datasets provide the fastest way to use PL/I to read and write data sets containing records of variable length and arbitrary byte patterns. This is not possible with TYPE(CRLF) data sets because when a record is read that was written containing the bit string '0d0a'b4, a misinterpretation occurs.

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

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

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

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

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

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

LLZZ
Specifies that records have a 4-byte prefix held the same way as varying records on S/390.

The LLZZ suboption provides a way to read and write data sets which contain records of variable length and arbitrary byte patterns which cannot be done with TYPE(CRLF) data sets. Under CRLF, a written record containing the bit string '0d0a'b4 is misinterpreted when it is read.

A TYPE(LLZZ) data set must not contain any record that is longer than the value determined for the record length of the data set.

CRLFEOF
Except for output files, this suboption specifies the same information as CRLF. When one of these files is closed for output, an end-of-file marker is appended to the last record.

U Indicates that records are unformatted. These unformatted files cannot be used by any record or stream I/O statements except OPEN and CLOSE. You can read from a TYPE(U) file only by using the FILEREAD built-in function. You can write to a TYPE(U) file only by using the FILEWRITE built-in function.

The TYPE option applies only to CONSECUTIVE files, except that it is ignored for printer-destined files with ASA(N) applied.

If your program attempts to access an existing data set with TYPE(FIXED) in effect and the length of the data set is not a multiple of the logical record length you specify, PL/I raises the UNDEFINEDFILE condition.

When using non-print files with the TYPE(FIXED) attribute, SKIP is replaced by trailing blanks to the end of the line. If TYPE(CRLF) is being used, SKIP is replaced by CRLF with no trailing blanks.

Associating a PL/I file with a data set

A file used within a PL/I program has a PL/I file name. A data set also has a name by which it is known to the operating system.

PL/I needs a way to recognize the data set(s) to which the PL/I files in your program refer, so you must provide an identification of the data set to be used, or allow PL/I to use a default identification.

You can identify the data set explicitly using either an environment variable or the TITLE option of the OPEN statement.

Using environment variables

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

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

```
declare MyFile stream output;
```

You can specify options for the SET command by including them on the command line.

```
set dd:myfile=c:\datapath\mydata.dat,APPEND(N)
```

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

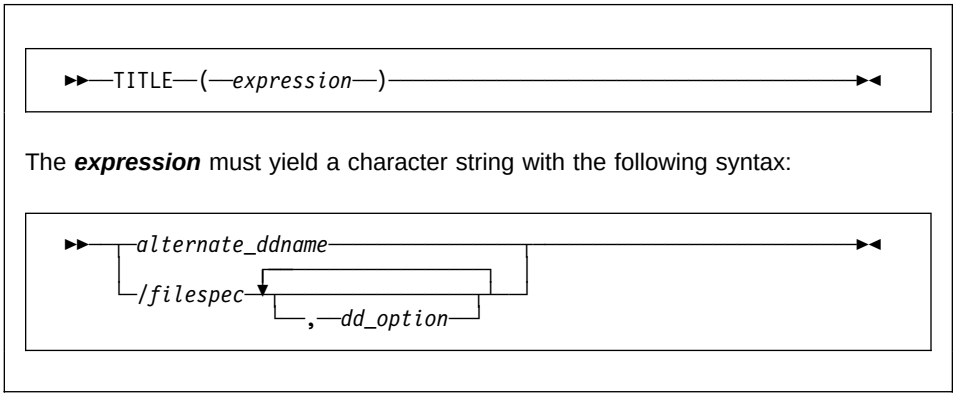
DD statement in MVS
ALLOCATE statement in TSO
FILEDEF command in CMS

For more about the syntax and options you can use with the DD:ddname environment variable, see “Specifying characteristics using DD:ddname environment variables” on page 215.

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.

Associating a PL/I file with a data set



alternate_ddname
The name of an alternate DD:ddname environment variable. An alternate DD:ddname environment variable is one not named after a file constant. For example, if you had a file named INVENTRY in your program, and you establish two DD:ddname environment variables—the first named INVENTRY and the second named PARTS—you could associate the file with the second one using this statement:

```
open file(Inventry) title('PARTS');
```

filespec
Any valid file specification on the system you are using.

dd_option
One or more options allowed in a DD:ddname environment variable. For more about options of the DD:ddname environment variable, see “Specifying characteristics using DD:ddname environment variables” on page 215.

Here is an example of using the OPEN statement in this manner:

```
open file(Payroll) title('/June.Dat,append(n),recsize(52)');
```

With this form, PL/I obtains all DD information either from the TITLE expression or from the ENVIRONMENT attribute of a file declaration. A DD:ddname environment variable is not referenced.

Attempting to use files not associated with data sets

If you attempt to use a file that has not been associated with a data set, (either through the use of the TITLE option of the OPEN statement or by establishing a DD:ddname environment variable), the UNDEFINEDFILE condition is raised. The only exceptions are the files SYSIN and SYSPRINT; these default to the CON device.

How PL/I finds data sets

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

- The current directory.
- The paths as defined in the DPATH environment variable.

Opening and closing PL/I files

This topic summarizes what PL/I does when your application executes the OPEN and CLOSE statements.

Opening a file

The execution of a PL/I OPEN statement associates a file with a data set. This requires merging of the information describing the file and the data set. The information is merged using the following order of precedence:

1. Attributes on the OPEN statement
2. ENVIRONMENT options on a file declaration
3. Values in TITLE option of the OPEN statement when '/' is used
4. Values in the DD:ddname environment variable
5. IBM defaults.

When the data set being opened is not a workstation device, the paths specified in the DPATH environment variable are searched for the data set. If the data set is not found, and the file has the OUTPUT attribute, the data set is created in the current directory.

If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Closing a file

The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated.

Associating several data sets with one file

A PL/I file can, at different times, represent entirely different data sets. The TITLE option allows you to choose dynamically, at open time, among several data sets to be associated with a particular PL/I file. Consider the following example:

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

In this example, when Master is opened during the first iteration of the do-group, the file is associated with the data set named MASTER1A.DAT. After processing, the file is closed, dissociating the PL/I file MASTER from the MASTER1A.DAT data set. During

the second iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the data set named MASTER1B.DAT. Similarly, during the final iteration of the do-group, MASTER is associated with the data set MASTER1C.DAT.

Combinations of I/O statements, attributes, and options

The figures that follow list the I/O statements, file attributes, ENVIRONMENT options, and DD:ddname environment variable options you can use for the various PL/I file operations. Table 13 on page 229 lists those for native data sets and Table 14 on page 230 lists those for workstation VSAM data sets.

Table 13. Statements, attributes, and options for native data sets

Statements	File attributes	ENVIRONMENT options	DD_DDNAME options
PUT	ENVIRONMENT FILE OUTPUT PRINT STREAM	CONSECUTIVE GRAPHIC RECSIZE(n)[1]	AMTHD(FSYS) APPEND(Y|N) ASA(Y|N)[2] file_spec RECSIZE(n)[1] SHARE(NONE|READ|ALL)[8] TERMLBUF(n)[3] TYPE(CRLF|TEXT|FIXED)
GET	ENVIRONMENT FILE STREAM INPUT	CONSECUTIVE GRAPHIC RECSIZE(n)[4]	AMTHD(FSYS) file_spec RECSIZE(n)[4] SHARE(NONE|READ|ALL)[8] TERMLBUF(n)[3] TYPE(CRLF|TEXT|FIXED)
WRITE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL[5] ENVIRONMENT FILE KEYED[6] RECORD OUTPUT|UPDATE	CONSECUTIVE|REGIONAL(1) CTLASA[78] RECSIZE(n)[1] SCALARVARYING[8]	AMTHD(FSYS) APPEND(Y|N) file_spec RECSIZE(n)[1] SHARE(NONE|READ|ALL)[8] TERMLBUF(n)[3] TYPE(CRLF|TEXT|FIXED)[7]
LOCATE[8]	BUFFERED ENVIRONMENT FILE KEYED[6] RECORD OUTPUT SEQUENTIAL	CONSECUTIVE|REGIONAL(1) CTLASA[2] RECSIZE(n)[1]	AMTHD(FSYS) APPEND(Y|N) file_spec RECSIZE(n)[1] SHARE(NONE|READ|ALL) TYPE(CRLF|TEXT|FIXED)[7]
READ	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL[5] ENVIRONMENT FILE INPUT|UPDATE KEYED[6] RECORD	CONSECUTIVE|REGIONAL(1) RECSIZE(n)[4] SCALARVARYING[8]	AMTHD(FSYS) file_spec RECSIZE(n)[4] SHARE(NONE|READ|ALL)[8] TERMLBUF(n)[3] TYPE(CRLF|TEXT|FIXED)[7]
REWRITE[8]	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL[5] ENVIRONMENT FILE UPDATE KEYED[6] RECORD	CONSECUTIVE|REGIONAL(1) RECSIZE(n)[4] SCALARVARYING	AMTHD(FSYS) file_spec RECSIZE(n)[4] SHARE(NONE|READ|ALL) TYPE(CRLF|TEXT|FIXED)[7]
DELETE[68]	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	REGIONAL(1) RECSIZE(n) SCALARVARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)

Notes:
1 When creating a new data set
2 When printer-destined PL/I file
3 When associated with a PM terminal
4 When data set was not created by PL/I program
5 DIRECT applicable only to REGIONAL(1)
6 For REGIONAL(1)
7 Not applicable to REGIONAL(1)
8 Not applicable to PM terminal (OS/2 only)

DISPLAY I/O

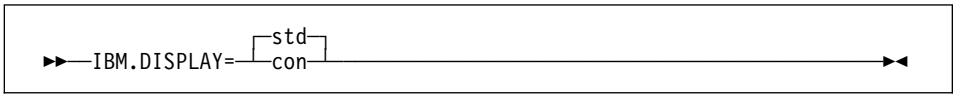
Table 14. Statements, attributes, and options for workstation VSAM data sets

Statements	File attributes	ENVIRONMENT options	DD_DDNAME options
PUT	ENVIRONMENT FILE OUTPUT PRINT STREAM	ORGANIZATION(CONSECUTIVE) GRAPHIC RECSIZE(n)[1]	AMTHD(DDM|ISAM|BTRIEVE) APPEND(Y|N) ASA(Y|N)[2] file_spec RECSIZE(n)[1] SHARE(NONE|READ|ALL)
GET	ENVIRONMENT FILE STREAM INPUT	ORGANIZATION(CONSECUTIVE) GRAPHIC RECSIZE(n)	AMTHD(DDM|ISAM|BTRIEVE) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
WRITE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE KEYED RECORD OUTPUT|UPDATE	ORGANIZATION|VSAM CTLASA RECSIZE(n) SCALARVARYING	AMTHD(DDM|ISAM|BTRIEVE) ASA(Y|N)[2] APPEND(Y|N)[3] file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
LOCATE	BUFFERED ENVIRONMENT FILE KEYED RECORD OUTPUT SEQUENTIAL	ORGANIZATION|VSAM CTLASA RECSIZE(n) SCALARVARYING	AMTHD(DDM|ISAM|BTRIEVE) APPEND(Y|N)[3] file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
READ	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE INPUT|UPDATE KEYED RECORD	ORGANIZATION|VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDM|ISAM|BTRIEVE) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
REWRITE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	ORGANIZATION|VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDM|ISAM|BTRIEVE) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)
DELETE	BUFFERED|UNBUFFERED DIRECT|SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	ORGANIZATION|VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDM|ISAM|BTRIEVE) file_spec RECSIZE(n) SHARE(NONE|READ|ALL)

Notes:
1 When creating a new data set
2 When printer-destined PL/I file
3 Does not apply to VSAM data sets

DISPLAY statement input and output

The REPLY in DISPLAY is read from stdin. Output from the DISPLAY statement is directed to stdout by default. The syntax of the IBM.DISPLAY environment variable is:



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

con
Specifies that the DISPLAY statement is to be associated with the CON device.

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

```
set ibm.display=std

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

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

```
hello > hello1.out
```

The greater than sign redirects the output to the file that is specified after it, in this case HELLO1.OUT. This means that the word 'HELLO' is written in the file HELLO1.OUT.

PL/I standard files (SYSPRINT and SYSIN)

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

Redirecting standard input, output, and error devices

You can also redirect standard input, standard output, and standard error devices to a file. You could use redirection in the following program, but you would first need to issue two SET DD: statements to allow the redirection to work. They are:

```
set dd:sysprint=stdout:
set dd:sysin=stdin:

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

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

```
hello > hello2.out
```

As is true with display statements, the greater than sign redirects the output to the file that is specified after it, in this case HELLO2.OUT. This means that the word 'HELLO' is written in the file HELLO2.OUT. Note also that the output includes printer control characters since the PRINT attribute is applied to SYSPRINT by default.

Redirecting devices

READ statements can access data from stdin, however, they must specify an LRECL equal to 1.

Chapter 12. Defining and using consecutive data sets

Printer-destined files

The sections that follow describe consecutive data set organization and explain how to create, access, and update consecutive data sets.

In a data set with consecutive organization, records are organized solely on the basis of their successive physical positions. In other words, when the data set is created, records are written consecutively in the order in which they are presented. You can retrieve the records only in the order in which they were written.

The information in this chapter applies to files using the CONSECUTIVE option of the ENVIRONMENT attribute that are associated with either a native or DDM data set. PL/I Presentation Manager supports only native data sets.

Printer-destined files

Printer-destined files are PL/I files with the PRINT attribute and record files declared with the CTLASA option of the ENVIRONMENT attribute. You can either print these files at your workstation or upload them to your mainframe.

The first character of each record is an American National Standard (ANS) carriage control character (see Table 15).

For STREAM files, PL/I inserts the character, based on the SKIP, LINE, or PAGE option (or control format item) of the PUT statement. For RECORD files with CTLASA, your program must insert the control characters in the first byte of each record.

If you want to print the data set from your workstation, select the ASA(N) option (it is the default). To keep the format for printing at the mainframe, select ASA(Y), which causes the control characters to be left untranslated.

Table 15. ANS print control characters

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

The translation to IBM Proprinter control characters is as follows:

Table 16. IBM Proprinter equivalents to ANS control characters

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

Note: Where:
0A = Line feed
0C = Form feed
0D = Carriage return

Only the first five characters listed are translated by PL/I; the others are treated as hyphens (-).

Using stream-oriented data transmission

This section covers how to define data sets for use with PL/I files that have the STREAM attribute. The essential parameters you use in the DD:ddname environment variable for creating and accessing these data sets are summarized, and several examples of PL/I programs are included.

Data sets with the STREAM attribute are processed by stream-oriented data transmission, which allows your PL/I program to ignore record boundaries and to treat a data set as a continuous stream of data values. Data values are either in character format or graphic format—that is, in DBCS (double byte character set) form. You create and access data sets for stream-oriented data transmission using the list-, data-, and edit-directed input and output statements described in the *PL/I Language Reference*.

For output, PL/I converts the data items from program variables into character format if necessary, and builds the stream of characters or DBCS characters into records for transmission to the data set. For input, PL/I takes records from the data set and separates them into the data items requested by your program, converting them into the appropriate form for assignment to program variables.

You can use stream-oriented data transmission to read or write DBCS data (graphics). DBCS data can be entered, displayed and printed if the appropriate devices have DBCS support. You must be sure that your data is in a format acceptable for the intended device or for a print utility program.

Stream-oriented transmission

Defining files using stream I/O

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

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

The FILE attribute is described in the *PL/I Language Reference*. The PRINT attribute is described further in “Using PRINT files” on page 241.

ENVIRONMENT options for stream-oriented data transmission

The ENVIRONMENT options you can use with stream-oriented data transmission are:

- CONSECUTIVE
- RECSIZE
- GRAPHIC
- ORGANIZATION(CONSECUTIVE).

You can find a description of these options and of their syntax in “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 209.

Creating a data set with stream I/O

To create a data set, use one of the following:

- ENVIRONMENT attribute
- DD:ddname environment variable
- TITLE option of the OPEN statement

Refer to “Using the TITLE option of the OPEN statement” on page 225 for more information on the TITLE option.

Essential information

When your application creates a STREAM file, it must supply a line size value for that file from one of the following sources:

- LINESIZE option of the OPEN statement
- RECSIZE option of the ENVIRONMENT attribute
- RECSIZE option of the TITLE option of the OPEN statement
- RECSIZE option of the DD:ddname environment variable
- PL/I-supplied default value

The PL/I default is used when you do not supply any value. If you choose the LINESIZE option, it overrides all other sources. The RECSIZE option of the ENVIRONMENT attribute overrides the other RECSIZE options. RECSIZE specified in the TITLE option of the OPEN statement has precedence over the RECSIZE option of the DD:ddname environment variable.

If LINESIZE is not supplied, but a RECSIZE value is, PL/I derives line size value from RECSIZE as follows:

- A PRINT file with the ASA(N) option applied has a RECSIZE value of 4
- A PRINT file with the ASA(Y) option applied has the RECSIZE value of 1
- Otherwise, the value of RECSIZE is assigned to the line size value.

PL/I determines a default line size value based on attributes of the file and the type of associated data set. In cases where PL/I cannot supply an appropriate default line size, the UNDEFINEDFILE condition is raised.

A default line size value is supplied for an OUTPUT file when:

- The file has the PRINT attribute. In this case, the value is obtained from the tab control table (see Figure 11 on page 244).
- The associated data set is the terminal (CON:, STDOUT:, or STDERR:). In this case the value is 120.

PL/I always derives the record length of the data set from the line size value. A record length value is derived from the line size value as follows:

- For a PRINT file, with the ASA(N) option applied, the value is line size + 4
- For a PRINT file, with the ASA(Y) option applied, the value is line size + 1
- Otherwise, the line size value is assigned to the record length value.

Example

Figure 8 on page 238 shows the use of stream-oriented data transmission to create a consecutive data set. The data is first read from the data set BDAY.INP that contains a list of names and birthdays of several people. Then a consecutive data set BDAY.OCT is written that contains the names and birthdays of people whose birthdays are in October.

The command SET DD:SYSIN=BDAY.INP should be used to associate the disk file BDAY.INP with the input data set. If this file was not created by a PL/I program, the RECSIZE option must also be specified.

The command SET DD:WORK=BDAY.OCT should be used to associate the consecutive output file WORK with the disk data set BDAY.OCT.

Stream-oriented transmission

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

BDAY: proc options(main);

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

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

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

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


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

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

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

Accessing a data set with stream I/O

It is not necessary that a data set accessed using stream-oriented data transmission was created by stream-oriented data transmission. However, it must have CONSECUTIVE organization, and all the data in it must be in character or graphic form. You can open the associated file for input, and read the items the data set contains; or you can open the file for output, and extend the data set by adding items at the end.

To access a data set, you must use one of the following to identify it:

- ENVIRONMENT attribute
- DD:ddname environment variable
- TITLE option of the OPEN statement

Essential information

When your application accesses an existing STREAM file, PL/I must obtain a record length value for that file. The value can come from one of the following sources:

- The LINESIZE option of the OPEN statement
- The RECSIZE option of the ENVIRONMENT attribute
- The RECSIZE option of the DD:ddname environment variable
- The RECSIZE option of the TITLE option of the OPEN statement
- An extended attribute of the data set
- PL/I-supplied default value.

If you are using an existing OUTPUT file, or if you supply a RECSIZE value, PL/I determines the record length value as described in “Creating a data set with stream I/O” on page 236.

PL/I uses a default record length value for an INPUT file when:

- The file is SYSIN, value = 80
- The file is associated with the terminal (CON:, SCREEN$:, STDOUT:, or STDERR:), value = 120.

Example

The program in Figure 9 on page 240 reads the data created by the program in Figure 8 on page 238 and uses the data set SYSPRINT to display that data. The SYSPRINT data set is associated with the OS/2 CON device, so if no dissociation is made prior to executing the program, the output is displayed on the screen. (For details on SYSPRINT, see “Using SYSIN and SYSPRINT files” on page 245.)

Stream-oriented transmission

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

BDAY1: proc options(main);

  dcl Work file stream input;

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

  dcl In char(30);

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

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

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

Using PRINT files

In a PL/I program, using a PRINT file provides a convenient means of controlling the layout of printed output from stream-oriented data transmission. PL/I automatically inserts print control characters in response to the PAGE, SKIP, and LINE options and format items.

You can apply the PRINT attribute to any STREAM OUTPUT file, even if you do not intend to print the associated data set directly. When a PRINT file is associated with a direct-access data set, the print control characters have no effect on the layout of the data set, but appear as part of the data in the records.

PL/I reserves the first byte of each record transmitted by a PRINT file for an American National Standard print control character, and inserts the appropriate characters automatically (see “Printer-destined files” on page 234).

PL/I handles the PAGE, SKIP, and LINE options or format items by inserting the appropriate control character in the records. If the SKIP or the LINE option specifies more than a 3-line space, PL/I inserts sufficient blank records with appropriate control characters to accomplish the required spacing.

If a PRINT file is being transmitted to a terminal device, the PAGE, SKIP, and LINE options never cause more than 3 lines to be skipped, unless formatted output is specified.

Controlling printed line length

You can limit the length of the printed line produced by a PRINT file by either:

- Specifying record length in your PL/I program using the RECSIZE option of the ENVIRONMENT attribute.
- Specifying line size in an OPEN statement using the LINESIZE option.
- Specifying record length in the TITLE option of the OPEN statement using the RECSIZE option.

RECSIZE must include the extra byte for the print control character; it must be 1 byte larger than the length of the printed line. LINESIZE refers to the number of characters in the printed line; PL/I adds the print control character.

Do not vary the line size for a file during execution by closing the file and opening it again with a new line size.

Since PRINT files have a default line size of 120 characters, you need not give any record length information for them.

Example: Figure 10 on page 242 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'.

Stream-oriented transmission

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

SINE: proc options(main);

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

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

Figure 10 (Part 1 of 2). Creating a print file via stream data transmission

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

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

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

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

Figure 10 (Part 2 of 2). Creating a print file via stream data transmission. (The example in Figure 15 on page 257 prints this file)

The statements in the ENDPAGE ON-unit insert a page number at the bottom of each page, and set up the headings for the following page.

The program in Figure 15 on page 257 uses record-oriented data transmission to print the table created by the program in Figure 10.

Overriding the tab control table

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions, which are defined in the PL/I-defined tab control table. The tab control table is an external structure named PLITABS. Figure 11 on page 244 shows its declaration.

Stream-oriented transmission

```
dcl 1 PLITABS static external,
  ( 2   Offset init (14),
    2    Pagesize init (60),
    2    Linesize init (120),
    2    Pagelength init (64),
    2    Fill1 init (0),
    2    Fill2 init (0),
    2    Fill3 init (0),
    2    Number_of_tabs init (5),
    2    Tab1 init (25),
    2    Tab2 init (49),
    2    Tab3 init (73),
    2    Tab4 init (97),
    2    Tab5 init (121)) fixed bin (15,0);
```

Figure 11. Declaration of PLITABS. (Gives standard page size, line size and tabulating positions)

The definitions of the fields in the table are as follows:

Offset
Binary integer that gives the offset of Number_of_tabs, the field that indicates the number of tabs to be used, from the top of PLITABS.

Pagesize
Binary integer that defines the default page size. This page size is used for dump output to the PLIDUMP data set as well as for stream output.

Linesize
Binary integer that defines the default line size.

Pagelength
Binary integer that defines the default page length for printing at a terminal. The value 0 indicates unformatted output.

Fill1, Fill2, Fill3
Three binary integers; reserved for future use.

Number_of_tabs
Binary integer that defines the number of tab position entries in the table (maximum 255). If tab count = 0, any specified tab positions are ignored.

Tab1—Tab*n*:
Binary integers that define the tab positions within the print line. The first position is numbered 1, and the highest position is numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it is ignored. The first data field in the printed output begins at the next available tab position.

You can override the default PL/I tab settings for your program by causing the linker to resolve an external reference to PLITABS. You do this by including a PL/I structure with the name PLITABS and the attributes EXTERNAL STATIC in the source program containing your main routine.

An example of the PL/I structure is shown in Figure 12 on page 245. 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 12. PL/I structure PLITABS for modifying the preset tab settings

Using SYSIN and SYSPRINT files

If you code GET or PUT statements without the FILE option, PL/I contextually assumes file SYSIN and SYSPRINT, respectively.

If you do not declare SYSPRINT, PL/I gives the file the attribute PRINT in addition to the normal default attributes; the complete set of attributes is:

```
file stream print external
```

Since SYSPRINT is a PRINT file, a default line size of 120 characters is applied when the file is opened.

You can override the attributes given to SYSPRINT by PL/I by explicitly declaring or opening the file. However, when SYSPRINT is declared or opened as a STREAM OUTPUT file, the PRINT attribute is applied by default unless the INTERNAL attribute is also declared.

PL/I does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes.

Controlling input from the console

To enter data for an input file, do both of the following:

- Declare the input file explicitly or implicitly with the CONSECUTIVE environment option (all stream files meet this condition)
- Allocate the input file to the terminal

Controlling input from the console

You can usually use the standard default input file SYSIN because it is a stream file and can be allocated to the console device.

You can be prompted for input to stream files by a colon (:) if you specify PROMPT(Y), see “PROMPT” on page 219. The colon is visible each time a GET statement is executed in the program. If you enter a line that does not contain enough data to complete execution of the GET statement, a further prompt is displayed. The GET statement causes the system to go to the next line. You can then enter the required data.

If you do not specify PROMPT(Y), the default is to have no colon visible at the beginning of the line.

By adding a hyphen to the end of any line that is to continue, you can delay transmission of the data to your program until you enter another line. The hyphen is an explicit continuation character.

If your program includes output statements that prompt for input, you can inhibit the initial system prompt by ending your own prompt with a colon. For example, the GET statement could be preceded by a PUT statement:

```
put skip list('Enter next item:');
```

To inhibit the system prompt for the next GET statement, your own prompt must meet the following conditions:

- It must be either list-directed or edit-directed, and if list-directed, must be to a PRINT file.
- The file transmitting the prompt must be allocated to the terminal. If you are using the COPY option to copy the file at the terminal, the system prompt is not inhibited.

Using files conversationally

To have your programs interact with a user conversationally, use the console as an input and output device for consecutive files in the program. Any stream file can be used conversationally, because conversational I/O needs no special PL/I code.

Format of data

The data you enter on the terminal should have exactly the same format as stream input data in batch mode, except for the following variations:

- *Simplified punctuation for input:* If you enter separate items of input on separate lines, there is no need to enter intervening blanks or commas; PL/I inserts a comma at the end of each line.

  As an example, consider the following statement:

  ```
  get list(I,J,K);
  ```

You could give the following response pressing the ENTER key after each item. (The colons only appear if you specify PROMPT(Y).

```
:
1
:
2
:
3
```

Entering the data on separate lines is equivalent to specifying:

```
:
1,2,3
```

If you wish to continue an item on another line, you must end the first line with a continuation character (the hyphen). Otherwise, for a GET LIST or GET DATA statement, a comma is inserted. For a GET EDIT statement, the item is padded.

- *Automatic padding for GET EDIT:* There is no need to enter blanks at the end of a line of input for a GET EDIT statement. The item you enter is padded to the correct length.

  Consider the following PL/I statement:

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

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

  ```
  SMITH
  ```

  The item is padded with 10 blanks, so that the program receives a string 15 characters long. If you wish to continue an item on a second or subsequent line, you must add a continuation character to the end of every line except the last. Otherwise, the first line transmitted would be padded and treated as the complete data item.

- *SKIP option or format item:* A SKIP in a GET statement ignores the data not yet entered. All uses of SKIP(n) where *n* is greater than one are taken to mean SKIP(1). SKIP(1) is taken to mean that all unused data on the current line is ignored.

Stream and record files

You can allocate both stream and record files to the terminal. However, no prompting is provided for record files. If you allocate more than one file to the terminal, and one or more of them is a record file, the output of the files is not necessarily synchronized. The order in which data is transmitted to and from the terminal is not guaranteed to be the same order in which the corresponding PL/I I/O statements are executed.

Capital and lowercase letters

For both stream and record files, character strings are transmitted to the program as entered in lowercase or uppercase.

Controlling output to the console

End of file

The characters /* in positions one and two of a line that contains no other characters are treated as an end-of-file mark and raise the ENDFILE condition.

Controlling output to the console

At your screen, you can display data from a PL/I file that has been both:

- Declared explicitly or implicitly with the CONSECUTIVE environment option. All stream files meet this condition.
- Allocated to the terminal device (CON:, STDOUT:, SCREEN$:, or STDERR:).

The standard output file SYSPRINT generally meets both these conditions.

Format of PRINT files

Data from SYSPRINT or other PRINT files is not normally formatted into columns and pages at the terminal. Three lines are always skipped for PAGE and LINE options and format items. The ENDPAGE condition is normally never raised. SKIP(n), where *n* is greater than three, causes only three lines to be skipped. SKIP(0) is implemented by carriage return.

You can cause a PRINT file to be formatted into pages by inserting a tab control table in your program. The table must be called PLITABS, and its contents are explained in "Overriding the tab control table" on page 243. For other than standard layout, use the information about PLITABS provided in Figure 11 on page 244. You can also use PLITABS to alter the tabulating positions of list-directed and data-directed output.

Tabulating of list-directed and data-directed output is achieved by transmission of blank (space) characters.

Stream and record files

You can allocate both stream and record files to the terminal. However, if you allocate more than one file to the terminal and one or more is a record file, the file output is not necessarily synchronized. There is no guarantee that the order in which data is transmitted between the program and the terminal is the same as the order in which the corresponding PL/I input and output statements are executed.

For stream and record files, characters are displayed on the terminal as they are held in the program. Both capital and lowercase characters can be displayed.

Example of an interactive program

The example program in Figure 13 on page 249 creates a consecutive data set PHONES using a dialog with the user. By default, SYSIN is associated with the CON device. You can override this association by setting an environment variable for the SYSIN file or by using the TITLE option on the OPEN statement. The output data set is associated with a disk file INT1.DAT and contains names and phone numbers that the user enters from the keyboard.

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

INT1: proc options(main);

  dcl Phones stream env(recsize(40));

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

  dcl 1 PhoneBookEntry,
       3 NameField char(19),
       3 PhoneNumber char(21);
  dcl InArea char(40);

  open file (Phones) output;

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

  /* start creating phone book */
  put list('Please enter name:');
  get edit(NameField)(a(19));
  if ¬Eof then
   do;
     put list('Please enter number:');
     get edit(PhoneNumber)(a(21));
   end;
  do while (¬Eof);
     put file(Phones) edit(PhoneBookEntry)(a(40));
     put list('Please enter name:');
     get edit(NameField)(a(19));
     if ¬Eof then
      do;
        put list('Please enter number:');
        get edit(PhoneNumber)(a(21));
      end;
  end;

  close file(Phones);

end INT1;
```

Figure 13. A sample interactive program

Using record-oriented I/O

PL/I supports various types of data sets with the RECORD attribute. This section covers how to use record-oriented I/O with consecutive data sets.

Table 17 lists the data transmission statements and options that you can use to create and access a consecutive data set using record-oriented I/O.

A CONSECUTIVE file that is associated with a DDM direct or keyed data set can be opened only for INPUT. PL/I raises UNDEFINEDFILE if an attempt is made to open such a file for OUTPUT or UPDATE.

Table 17 (Page 1 of 2). Statements and options allowed for creating and accessing consecutive data sets

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

Table 17 (Page 2 of 2). 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 UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	

Notes:

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

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

Defining files using record I/O

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

```
declare
  Filename file record
           input | output | update
           sequential
           buffered | unbuffered
           environment(options);
```

The file attributes are described in the *PL/I Language Reference*.

ENVIRONMENT options for record-oriented data transmission

The ENVIRONMENT options applicable to consecutive data sets for record-oriented data transmission are:

- CONSECUTIVE
- CTLASA
- ORGANIZATION(CONSECUTIVE)
- RECSIZE
- SCALARVARYING

You can find a description of these options and of their syntax in “Specifying characteristics using the PL/I ENVIRONMENT attribute” on page 209.

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 17 on page 250 shows the statements and options for creating a consecutive data set.

Using record-oriented I/O

To create a data set, you must give PL/I certain information either in the ENVIRONMENT attribute, in a DD:ddname environment variable, or in the TITLE option of the OPEN statement.

Essential information

When you create a consecutive data set you must specify:

- The name of data set to be associated with your PL/I file. A data set with consecutive organization can exist on any type of device (see “Attempting to use files not associated with data sets” on page 226).
- The record length. You can specify the record length using the RECSIZE option of the ENVIRONMENT attribute, of the DD:ddname environment variable, or of the TITLE option of the OPEN statement.

  For files associated with the terminal device (CON:, STDOUT:, or STDERR:), PL/I uses a default record length of 120 when the RECSIZE option is not specified.

Accessing and updating a data set with record I/O

Once you create a consecutive data set, you can open the file that accesses it for sequential input, for sequential output, or, for data sets on direct-access devices, for updating. For an example of a program that accesses and updates a consecutive data set, see Figure 14 on page 254.

If you open the file for output, and wish to extend the data set by adding records at the end, you need not specify APPEND(Y) in the DD:ddname environment variable, since this is the default. If you specify APPEND(N), the data set is overwritten. If you open a file for updating, you can only update records in their existing sequence, and if you want to insert records, you must create a new data set. You cannot change the record length of an existing data set.

When you access a consecutive data set by a SEQUENTIAL UPDATE file, you must retrieve a record with a READ statement before you can update it with a REWRITE statement. Every record that is retrieved, however, need not be rewritten. A REWRITE statement always updates the last record read.

Consider the following:

```
read file(F) into(A);
       ⋮
read file(F) into(B);
       ⋮
rewrite file(F) from(A);
```

The REWRITE statement updates the record that was read by the second READ statement. The record that was read by the first statement cannot be rewritten after the second READ statement has been executed.

To access a data set, you must identify it to PL/I using the TITLE option of the OPEN statement or a DD:ddname environment variable.

Table 17 on page 250 shows the statements and options for accessing and updating a consecutive data set.

Essential information

When your application accesses an existing RECORD file, PL/I must obtain a record length value for that file. The value can come from one of the following sources:

- The RECSIZE option of the ENVIRONMENT attribute
- The RECSIZE option of the DD:ddname environment variable
- The RECSIZE option of the TITLE option of the OPEN statement
- PL/I-supplied default value.

PL/I uses a default record length value for an INPUT file when:

- The file is SYSIN. In this case, the value used is 80.
- The file is associated with the terminal. In this case, the value used is 120.

Examples of consecutive data sets

Creating and accessing consecutive data sets are illustrated in the program in Figure 14 on page 254. The program merges the contents of two PL/I files INPUT1 and INPUT2, and writes them onto a new PL/I file, OUT. INPUT1 and INPUT2 are associated with the disk files EVENS.INP and ODDS.INP, respectively, and contain 6-byte records arranged in ASCII collating sequence.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Merge 2 data sets creating a CONSECUTIVE data set.             */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      SET DD:OUT=CON4.DAT                                          */
/*      SET DD:INPUT1=EVENS.INP                                      */
/*      SET DD:INPUT2=ODDS.INP                                       */
/*                                                                   */
/*********************************************************************/

 MERGE: proc options(main);

   dcl Input1 file record sequential input env(recsize(6));
   dcl Input2 file record sequential input env(recsize(6));
   dcl Out    file record sequential env(recsize(15));
   dcl Sysprint    file print;              /* normal print file    */

   dcl Input1_Eof  bit(1) init('0'b);       /* eof flag for Input1  */
   dcl Input2_Eof  bit(1) init('0'b);       /* eof flag for Input2  */
   dcl Out_Eof     bit(1) init('0'b);       /* eof flag for Out     */
   dcl True        bit(1) init('1'b);       /* constant True        */
   dcl False       bit(1) init('0'b);       /* constant False       */

   dcl Item1       char(6) based(a);        /* item from Input1     */
   dcl Item2       char(6) based(b);        /* item from Input2     */
   dcl A           pointer;                 /* pointer var          */
   dcl B           pointer;                 /* pointer var          */

   on endfile(Input1) Input1_Eof = True;
   on endfile(Input2) Input2_Eof = True;
   on endfile(Out)    Out_Eof    = True;

   open file(Input1),
        file(Input2),
        file(Out) output;

   read file(Input1) set(A);                /* priming read         */
   read file(Input2) set(B);
```

Figure 14 (Part 1 of 3). Merge Sort—Creating and accessing a consecutive data set

```
    do while ((Input1_Eof = False) & (Input2_Eof = False));
      if Item1 > Item2 then
        do;
          write file(Out) from(Item2);
          put file(Sysprint) skip edit('1>2', Item1, Item2)
              (a(5),a,a);
          read file(Input2) set(B);
        end;
      else
        do;
          write file(Out) from(Item1);
          put file(Sysprint) skip edit('1<2', Item1, Item2)
              (a(5),a,a);
          read file(Input1) set(A);
        end;
    end;

    do while (Input1_Eof = False);           /* Input2 is exhausted  */
      write file(Out) from(Item1);
      put file(Sysprint) skip edit('1', Item1) (a(2),a);
      read file(Input1) set(A);
    end;

    do while (Input2_Eof = False);           /* Input1 is exhausted  */
      write file(Out) from(Item2);
      put file(Sysprint) skip edit('2', Item2) (a(2),a);
      read file(Input2) set(B);
    end;

    close file(Input1), file(Input2), file(Out);
    put file(Sysprint) page;
    open file(Out) sequential input;

    read file(Out) into(Item1);              /* display Out file     */
    do while (Out_Eof = False);
      put file(Sysprint) skip edit(Item1) (a);
      read file(Out) into(Item1);
    end;
    close file(Out);

  end MERGE;
```

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

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

BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ

Here is a sample of ODDS.INP:

AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
KKKKKK
```

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

The program in Figure 15 uses record-oriented data transmission to print the table created by the program in Figure 10 on page 242.

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

PRT: proc options(main);

  dcl Table       file record input sequential;
  dcl Printer     file record output seql
                       env(recsize(200) ctlasa);
  dcl Line        char(102) var;

  dcl Table_Eof   bit(1) init('0'b);       /* Eof flag for Table   */
  dcl True        bit(1) init('1'b);       /* constant True        */
  dcl False       bit(1) init('0'b);       /* constant False       */


  on endfile(Table) Table_Eof = True;

  open file(Table),
       file(Printer);

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

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

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

Figure 15. Printing record-oriented data transmission

Chapter 13. Defining and using regional data sets

Regional data sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. Creating and accessing regional data sets are also discussed.

A data set with regional organization is divided into regions, each of which is identified by a region number, and each of which can contain one record. The regions are numbered in succession, beginning with zero, and a record can be accessed by specifying its region number in a data transmission statement.

Regional data sets are confined to direct-access devices.

Regional organization of a data set allows you to control the physical placement of records in the data set and to optimize the data access time. This type of optimization is not available with consecutive organization, in which successive records are written in strict physical sequence.

You can create a regional data set in a manner similar to a consecutive data set, presenting records in the order of ascending region numbers; alternatively, you can use direct-access, in which you present records in random sequence and insert them directly into preformatted regions. Once you create a regional data set, you can access it by using a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. You do not need to specify either a region number or a key if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Records within a regional data set are either actual records containing valid data or dummy records.

PL/I supports REGIONAL(1) data sets. See Table 18 on page 260 for a list of the data transmission statements and options that you can use to create and access a REGIONAL(1) data set.

Regional data sets

Table 18 (Page 1 of 2). Statements and options allowed for creating and accessing regional data sets

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

Table 18 (Page 2 of 2). 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
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 cannot have the UPDATE attribute when creating new data sets.

Defining files for a regional data set

Use a file declaration with the following attributes to define a sequential regional data set:

```
declare
  Filename file record
           input | output | update
           sequential
           buffered | unbuffered
          [keyed]
           environment(options);
```

Using REGIONAL(1) data sets

To define a direct regional data set, use a file declaration with the following attributes:

```
declare
  Filename file record
           input | output | update
           direct
           unbuffered
          [keyed]
           environment(options);
```

File attributes are described in the *PL/I Language Reference*.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to regional data sets are:

REGIONAL(1)
RECSIZE
SCALARVARYING

These options are described in "Specifying characteristics using the PL/I ENVIRONMENT attribute" on page 209.

Essential information for creating and accessing regional data sets

To create a regional data set, you must give PL/I certain information, either in the ENVIRONMENT attribute or in the DD:ddname environment variable.

You must supply the following information when creating a regional data set:

- The name of the data set associated with your PL/I file. A data set with REGIONAL(1) organization can exist only on a direct-access storage device (see "Attempting to use files not associated with data sets" on page 226).
- The record length. You can specify the record length using the RECSIZE option of the ENVIRONMENT attribute or of the DD:ddname environment variable or in the TITLE option of the OPEN statement.
- The extent (the number of regions) of the data set. You specify this with the RECCOUNT option of the DD:ddname environment variable.

  The default for RECCOUNT is 50.

Using keys with regional data sets

Source keys are used to access REGIONAL(1) data sets. A *source key* is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. When you access a record in a regional data set, the source key is the region number.

Using REGIONAL(1) data sets

In a REGIONAL(1) data set, the region number serves as the sole identification of a particular record. The character value of the source key should represent an unsigned

decimal integer that should not exceed 2147483647. If the region number exceeds this figure, it is treated as modulo 2147483648; for instance, 2147483658 is treated as 10.

Only the characters 0 through 9 and the blank character are valid in the source key; leading blanks are interpreted as zeros. Embedded blanks are not allowed in the region number; the first embedded blank, if any, terminates the region number. If more than 10 characters appear in the source key, only the rightmost 10 are used as the region number; if there are fewer than 10 characters, blanks (interpreted as zeros) are inserted on the left.

Dummy records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant X'FF' in its first byte. Although such dummy records are inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read. Your PL/I program must be prepared to recognize them. You can replace dummy records with valid data.

Creating a REGIONAL(1) data set

You can create a REGIONAL(1) data set either sequentially or by direct-access. Table 18 on page 260 shows the statements and options for creating a regional data set.

When you create the data set, opening the file causes the data set to be filled with dummy records. You must present records in ascending order of region numbers for a SEQUENTIAL OUTPUT file. If there is an error in the sequence, or if you present a duplicate key, the KEY condition is raised. If you use a DIRECT OUTPUT file to create the data set, you can present records in random order. If you present a duplicate region number, the existing record is overwritten.

If you create a data set using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement might raise the ERROR condition.

Example

Creating a REGIONAL(1) data set is illustrated in Figure 16 on page 264. The data set is a list of telephone extensions with the names of the subscribers to whom they are allocated. The telephone extensions correspond with the region numbers in the data set; the data in each occupied region being a subscriber's name.

Using REGIONAL(1) data sets

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Create a REGIONAL(1) data set.                                 */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*     SET DD:SYSIN=CRG.INP,RECSIZE(30)                              */
/*     SET DD:NOS=NOS.DAT,RECCOUNT(100)                              */
/*                                                                   */
/*********************************************************************/

 CRR1: proc options(main);

   dcl  Nos file record output direct keyed
        env(regional(1) recsize(20));

   dcl  Sysin file input record;
   dcl 1   In_Area,
        2   Name   char(20),
        2   Number char( 2);
   dcl IoField char(20);
   dcl Sysin_Eof bit (1) init('0'b);
   dcl Ntemp fixed(15);

      on endfile (Sysin) Sysin_Eof = '1'b;

      open file(Nos);
      read file(Sysin) into(In_Area);
      do while(¬Sysin_Eof);
         IoField = Name;
         Ntemp = Number;
         write file(Nos) from(IoField) keyfrom(Ntemp);
         put file(sysprint) skip edit (In_Area) (a);
         read file(Sysin) into(In_Area);
      end;

      close file(Nos);
  end CRR1;
```

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

```
The execution time input file, CRG.INP, might look like this:

ACTION,G.           12
BAKER,R.            13
BRAMLEY,O.H.        28
CHEESNAME,L.        11
CORY,G.             36
ELLIOTT,D.          85
FIGGINS,E.S.        43
HARVEY,C.D.W.       25
HASTINGS,G.M.       31
KENDALL,J.G.        24
LANCASTER,W.R.      64
MILES,R.            23
NEWMAN,M.W.         40
PITT,W.H.           55
ROLF,D.E.           14
SHEERS,C.D.         21
SURCLIFFE,M.        42
TAYLOR,G.C.         47
WILTON,L.W.         44
WINSTONE,E.M.       37
```

Figure 16 (Part 2 of 2). 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 18 on page 260 shows the statements and options for accessing a regional data set.

Sequential access

To open a SEQUENTIAL file that is used to process a REGIONAL(1) data set, use either the INPUT or UPDATE attribute. You must not include the KEY option in data transmission statements, but the file can have the KEYED attribute, since you can use the KEYTO option. If the target character string referenced in the KEYTO option has more than 10 characters, the value returned (the 10-character region number) is padded on the left with blanks. If the target string has fewer than 10 characters, the value returned is truncated on the left.

Sequential access is in the order of ascending region numbers. All records are retrieved, whether dummy or actual, and you must ensure that your PL/I program recognizes dummy records.

Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region-number sequence, and in sequential update you can read and rewrite each record in turn.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(1) data set are identical to

those for a consecutive data set. A discussion of using READ and REWRITE statements can be found in “Accessing and updating a data set with record I/O” on page 252.

Direct access

To open a DIRECT file that is used to process a REGIONAL(1) data set you can use either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

Use DIRECT UPDATE files to retrieve, add, delete, or replace records in a REGIONAL(1) data set according to the following conventions:

Retrieval All records, whether dummy or actual, are retrieved. Your program must recognize dummy records.

Addition A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key.

Deletion The record you specify by the source key in a DELETE statement is turned into a dummy record.

Replacement The record you specify by the source key in a REWRITE statement, whether dummy or actual, is replaced.

Example

Updating a REGIONAL(1) data set is illustrated in Figure 17 on page 267. This program updates the data set and lists its contents. Before each new or updated record is written, the existing record in the region is tested to ensure that it is a dummy. This is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Update a REGIONAL(1) data set.                                 */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      SET DD:SYSIN=ACR.INP,RECSIZE(30)                             */
/*      SET DD:NOS=NOS.DAT,APPEND(Y)                                 */
/*                                                                   */
/*    Note: This sample program is using the regional dataset,       */
/*          NOS.DAT, created by the previous sample program CRR1.    */
/*                                                                   */
/*********************************************************************/

 ACR1: proc options(main);

   dcl Nos file record  keyed env(regional(1));
   dcl  Sysin file input record;
   dcl Sysin_Eof bit (1) init('0'b);
   dcl   Nos_Eof bit (1) init('0'b);
   dcl 1   In_Area,
         2   Name   char(20),
         2   (CNewNo,COldNo)  char( 2),
         2   In_Area_1 char( 1),
         2   Code   char( 1);
   dcl IoField char(20);
   dcl Byte    char(1) def IoField;
   dcl NewNo fixed(15);
   dcl OldNo fixed(15);

   on endfile (Sysin) Sysin_Eof = '1'b;
   open file (Nos) direct update;
   read file(Sysin) into(In_Area);
```

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

Using REGIONAL(1) data sets

```
   do while(¬Sysin_Eof);
      if CNewNo ¬='  ' then
         NewNo = CNewNo;
      else
         NewNo = 0;
      if COldNo ¬='  ' then
         OldNo = COldNo;
      else
         OldNo = 0;
      select(Code);
         when('A','C')
         do;
           if Code = 'C' then
              delete file(Nos) key(OldNo);
           read file(Nos) key(NewNo) into(IoField);
             /*  we must test to see if the record exists  */
             /* if it doesn't exist we create a record there  */
           if unspec(Byte) = (8)'1'b then
              write file(Nos) keyfrom(NewNo) from(Name);
           else put file(sysprint) skip list ('duplicate:',Name);
         end;
         when('D') delete file(Nos) key(OldNo);
         otherwise put file(sysprint) skip list ('invalid code:',Name);
      end;
      read file(Sysin) into(In_Area);
   end;

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

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

```
At execution time, the input file, ACR.INP, could look like this:

NEWMAN,M.W.          5640 C
GOODFELLOW,D.T.      89   A
MILES,R.               23 D
HARVEY,C.D.W.        29   A
BARTLETT,S.G.        13   A
CORY,G.                36 D
READ,K.M.            01   A
PITT,W.H.              55 X
ROLF,D.F.              14 D
ELLIOTT,D.           4285 C
HASTINGS,G.M.          31 D
BRAMLEY,O.H.         4928 C
```

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

Chapter 14. Defining and using workstation VSAM data sets

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

Platform distinction

Three access methods are discussed in connection with the PL/I workstation products; however, not all three methods are supported on every platform. Use the following as a guideline:

- DDM—supported on AIX and OS/2 only
- ISAM—supported on AIX, OS/2, and Windows
- BTRIEVE—supported on OS/2 and Windows only

This chapter also describes the statements you use to access the three types of VSAM data sets—sequential, keyed, and direct. In many ways, workstation VSAM is similar to the VSAM on the mainframe. On the workstation, the terms sequential, keyed and direct are similar to the VSAM entry-sequenced data set, key-sequenced data set, and relative record data set.

The chapter concludes with a series of examples showing the PL/I statements and DD:ddname environment variables necessary to create and access workstation VSAM data sets.

Moving data between the workstation and mainframe

To convert mainframe VSAM files to the corresponding DDM, ISAM, or BTRIEVE files, follow the procedure documented in the prolog for the LODVSAM utility. Make sure you specify the appropriate access method AMTHD(DDM|ISAM|BTRIEVE).

To convert DDM, ISAM, or BTRIEVE files to corresponding mainframe VSAM files, follow the procedure documented in the prolog for the RELOAD utility. These utilities are supported on VisualAge PL/I, but are not currently available on PL/I Set for AIX.

Workstation VSAM organization

PL/I supports workstation VSAM sequential, keyed, and direct data sets. These correspond to PL/I consecutive, indexed, and relative data set organizations, respectively.

Both sequential and keyed access are possible with all three types of data sets. With keyed data sets, the key, which is part of the logical record, is used for keyed access; keyed access is possible for direct data sets using relative record numbers. Keyed access is also possible for sequential data sets using the sequential record value as a key.

All workstation VSAM data sets are stored on direct-access storage devices. The physical organization of workstation VSAM data sets differs from those used by other access methods.

Workstation VSAM organization

Creating and accessing workstation VSAM data sets

Your PL/I application can create workstation VSAM data sets, or it can access VSAM data sets created by other programs. When you open a file to be associated with a workstation VSAM data set, and that data set does not exist, PL/I creates it using the attributes and options you specify in the DECLARE statement or in a DD:ddname environment variable.

When your application accesses an existing VSAM data set, PL/I determines its type—sequential, direct, or keyed.

The operation of writing the initial data into a newly-created VSAM data set is referred to as *loading* in this publication.

Are you using the right access method?

Use each of the access methods, DDM|ISAM|BTRIEVE, to access data sets that were created with that particular access method. For example, you cannot use the ISAM access method to access data sets you created with the BTRIEVE access method.

Determining which type of workstation VSAM data set you need

Use the three different types of data sets according to the following purposes:

- Use *sequential data sets* for data that you access primarily in the order in which the records were created (or the reverse order).
- Use *keyed data sets* when you normally access records through keys within the records (for example, a stock-control file where the part number is used to access a record).
- Use *direct data sets* for data in which each item has a particular number, and you normally access the relevant record by that number (for example, a telephone system with a record associated with each number).

Accessing records in workstation VSAM data sets

You can access records in all types of workstation VSAM data sets either directly by means of a key or sequentially (backward or forward). You can also use a combination of the two ways, in which you select a starting point with a key and then read forward or backward from that point.

Table 19 on page 273 shows how data could be stored in the three different types of workstation VSAM data sets and illustrates their respective advantages and disadvantages.

Table 19. Types and advantages of workstation VSAM data sets

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

Using keys for workstation VSAM data sets

All workstation VSAM data sets can have keys associated with their records. For keyed data sets, the key is a defined field within the logical record. For sequential data sets, the key is the sequential record value of the record. For relative record data sets, the key is a *relative record number*.

Using keys for workstation VSAM keyed data sets

Keys for keyed data sets are part of the logical records recorded on the data set. You define the length and location of the keys when you create the data set.

Choosing a data set type

The ways you can reference the keys in the KEY, KEYFROM, and KEYTO options are as described under “KEY(expression) Option,” “KEYFROM(expression) Option,” and “KEYTO(reference) Option” in the *PL/I Language Reference*.

Using sequential record values

Sequential record values allow you to use keyed access on a sequential data set associated with a KEYED SEQUENTIAL file.

DDM (OS/2 only)
The sequential record values, or keys, are character strings of length 4, and their values are defined by workstation VSAM.

BTRIEVE and ISAM
The sequential record values, or keys, are character strings of length 7, and their values are defined by workstation VSAM.

You cannot construct or manipulate sequential record values in PL/I; you can, however, compare their values in order to determine the relative positions of records within the data set. Sequential record values are not normally printable.

You can obtain the sequential record value for a record by using the KEYTO option, either on a WRITE statement when you are loading or extending the data set, or on a READ statement when the data set is being read. You can subsequently use a sequential record value obtained in either of these ways in the KEY option of a READ or REWRITE statement.

Using relative record numbers

Records in a direct data set are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record. You can use these relative record numbers as keys for keyed access to the data set.

Keys used as relative record numbers are character strings of length 10. The character value of a source key you use in the KEY or KEYFROM option must represent an unsigned integer. If the source key is not 10 characters long, it is truncated or padded with blanks (interpreted as zeros) on the **left**. The value returned by the KEYTO option is a character string of length 10, with leading zeros suppressed.

Choosing a data set type

When planning your application, you must first decide which type of data set to use. There are three types of workstation VSAM data sets available to you. Workstation VSAM data sets can provide all the function of the other types of data sets, plus additional function available only with workstation VSAM. Workstation VSAM can usually match, or even improve upon, the performance of other data set types. However, workstation VSAM is more subject to performance degradation through misuse of function.

Table 19 on page 273 shows you the possibilities available with each type of workstation VSAM data set. When choosing between the workstation VSAM data set

types, you should base your decision on the most common sequence in which your program accesses your data.

Table 20 on page 278, Table 21 on page 282, and Table 22 on page 292 show the statements allowed for sequential data sets, keyed data sets, and direct data sets, respectively.

Defining files for workstation VSAM data sets

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

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

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

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

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

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

The file attributes are described in the *PL/I Language Reference* for this product. Options of the ENVIRONMENT attribute are discussed below.

Specifying options of the PL/I ENVIRONMENT attribute

Many of the options of the PL/I ENVIRONMENT attribute affecting data set structure are not needed for workstation VSAM data sets. If you specify them, they are either ignored or are used for checking purposes. If those that are checked conflict with the values defined for the data set, the UNDEFINEDFILE condition is raised when an attempt is made to open the file.

The ENVIRONMENT options applicable to workstation VSAM data sets are:

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

For a complete explanation of these ENVIRONMENT options and how to use them, see "Specifying characteristics using the PL/I ENVIRONMENT attribute" on page 209. In addition to this list of ENVIRONMENT options, there is a set of options that can be used with a DD statement, see "Specifying characteristics using DD:ddname environment variables" on page 215.

Adapting existing programs for workstation VSAM

This section is intended primarily for OS PL/I users who are transferring programs to the workstation.

In most cases, if your PL/I program uses files declared with ENVIRONMENT (CONSECUTIVE) or ENVIRONMENT(INDEXED) or with no PL/I ENVIRONMENT attribute, it can access workstation VSAM data sets without alteration. PL/I detects that a workstation VSAM data set is being opened and can provide the correct access.

You can readily adapt existing programs with CONSECUTIVE, INDEXED, REGIONAL(1) or VSAM files for use with workstation VSAM data sets. Programs with consecutive files might not need alteration, and there is never any necessity to alter programs with indexed files unless the logic depends on EXCLUSIVE files. Programs with REGIONAL(1) data sets require only minor revision.

The following sections tell you what modifications you might need to make in order to adapt files for the workstation.

Adapting programs using CONSECUTIVE files

There is no concept of fixed-length records in DDM, but there is in ISAM and BTRIEVE. If your program relies on the RECORD condition to detect incorrect length records, it does not function in the same way using workstation VSAM data sets as it does with non-workstation VSAM data sets.

If the logic of the program depends on raising the RECORD condition when a record of an incorrect length is found, you must write your own code to check for the record length and take the necessary action. This is because records of any length up to the maximum specified are allowed in workstation VSAM data sets.

Adapting programs using INDEXED files

Compatibility is provided for INDEXED files. For files that you declare with the INDEXED ENVIRONMENT option, PL/I associates the file with a workstation VSAM keyed data set. UNDEFINEDFILE is raised if the data set is any other type.

Because mainframe ISAM record handling differs in detail from workstation VSAM record handling, workstation VSAM processing might not always give the required result.

You should remove dependence on the RECORD condition, and insert your own code to check for record length if this is necessary. You should also remove any checking for deleted records.

Adapting programs using REGIONAL(1) files

You can alter programs using REGIONAL(1) data sets to use workstation VSAM direct data sets. Remove REGIONAL(1) and any other implementation-dependent options from the file declaration and replace them with ENV(ORGANIZATION(RELATIVE)). You should also remove any checking for deleted records, because workstation VSAM deleted records are not accessible to you.

Adapting programs using VSAM files

If you use the VSAM ENVIRONMENT option, the associated workstation VSAM data set must exist before the file is opened. You can create your data sets with a simple program. Figure 18 is an example of creating a workstation VSAM keyed data set.

```
/*********************************************************************/
/*                                                                   */
/*  NAME - ISAM0.PLI                                                 */
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Create an ISAM Keyed data set                                  */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

NewVSAM: proc options(main);
     declare
         NewFile keyed record output file
                 env(organization(indexed)
                     recsize(80)
                     keylength(8)
                     keyloc(17)
                    );
         open file(NewFile) title('/KEYNAMES.DAT');
         close file(NewFile);
End NewVSAM;
```

Figure 18. Creating a workstation VSAM keyed data set

If the data set named KEYNAMES.DAT does not already exist, PL/I creates it with that name when the OPEN statement is executed.

Using workstation VSAM sequential data sets

The statements and options allowed for files associated with a workstation VSAM sequential data set are shown in Table 20.

Table 20 (Page 1 of 2). Statements and options allowed for loading and accessing workstation VSAM sequential data sets

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

Table 20 (Page 2 of 2). Statements and options allowed for loading and accessing workstation VSAM sequential data sets

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

Notes:

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

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

[3] The expression used in the KEY option must be a sequential record value, previously obtained by means of the KEYTO option.

Using a sequential file to access a workstation VSAM sequential data set

When a sequential data set is being loaded, the associated file must be opened for SEQUENTIAL OUTPUT. The records are stored in the order in which they are presented.

You can use the KEYTO option to obtain the sequential record value of each record as it is written. You can subsequently use these keys to achieve keyed access to the data set.

You can open a SEQUENTIAL file that is used to access a workstation VSAM sequential data set with either the INPUT or the UPDATE attribute. If you use either of the options KEY or KEYTO, the file must also have the KEYED attribute.

Sequential access occurs in the order that the records were originally loaded into the data set. You can use the KEYTO option on the READ statements to recover the sequential record value of the records that are read. If you use the KEY option, the record that is recovered is the one with the sequential record value you specify. Subsequent sequential access continues from the new position in the data set.

For an UPDATE file, the WRITE statement adds a new record at the end of the data set. With a REWRITE statement, the record rewritten is the one with the specified sequential record value if you use the KEY option; otherwise, it is the record accessed on the previous READ.

Using workstation VSAM sequential data sets

Defining and loading a workstation VSAM sequential data set

Figure 19 on page 281 is an example of a program that defines and loads a workstation VSAM sequential data set.

The PL/I program writes the data set using a SEQUENTIAL OUTPUT file and a WRITE FROM statement.

The sequential record values of the records could have been obtained during the writing for subsequent use as keys in a KEYED file. To do this, a suitable variable would have to be declared to hold the key and the WRITE...KEYTO statement used. For example:

```
dcl Chars char(7); /*DDM uses 4; BTRIEVE and ISAM use 7 as shown */
write file(Famfile) from (String)
  keyto(Chars);
```

The keys would not normally be printable, but could be retained for subsequent use.

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

 CREATE: proc options(main);

    dcl
      FamFile file sequential output
                  env(organization(consecutive)),
      In file record input,
      Eof bit(1) init('0'b),
      i   fixed(15),
      String char(38);

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

    read file(In) into (String);
    do i=1 by 1 while (¬Eof);
      put file(sysprint) skip edit (String) (a);
      write file(FamFile) from (String);
      read file(In) into (String);
    end;

    put skip edit(i-1,' records processed ')(a);
  end CREATE;

The input data for this program might look like this:

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

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

Updating a sequential data set

The program illustrated in Figure 19 can be used to update a workstation VSAM sequential data set. If it is run again, new records are added on the end of the data set.

You can rewrite existing records in a sequential data set, provided that the length of the record is not changed. You can use a SEQUENTIAL or KEYED SEQUENTIAL update

file to do this. If you use keys, they must be sequential record values from a previous WRITE or READ statement.

Workstation VSAM keyed data sets

The statements and options allowed for workstation VSAM keyed data sets are shown in Table 21.

Table 21 (Page 1 of 4). Statements and options allowed for loading and accessing workstation VSAM keyed data sets

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

Table 21 (Page 2 of 4). Statements and options allowed for loading and accessing workstation VSAM keyed data sets

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

Workstation VSAM keyed data sets

Table 21 (Page 3 of 4). Statements and options allowed for loading and accessing workstation VSAM keyed data sets

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

Table 21 (Page 4 of 4). Statements and options allowed for loading and accessing workstation VSAM keyed data sets

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

Notes:

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

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

[3] Do not associate a DIRECT file with a workstation VSAM data set that has duplicate key capability.

Workstation VSAM keyed data sets

Loading a workstation VSAM keyed data set

When a keyed data set is being loaded, you must open the associated file for KEYED SEQUENTIAL OUTPUT. You must present the records in ascending key order, and you must use the KEYFROM option.

If a keyed data set already contains some records, and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can add records at the end of the data set only. Again, you must present the records in ascending key order, and you must use the KEYFROM option. In addition, the first record you present must have a key greater than the highest key present on the data set.

Figure 20 on page 287 is an example of a program that loads a workstation VSAM keyed data set. Within the PL/I program, a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A keyed data set must be loaded in this manner.

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

NAMELD: proc options(main);

  dcl Direc file record keyed sequential output
      env(organization(indexed)
          recsize(23)
          keyloc(1)
          keylength(20)
         );

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

  dcl 1 IoArea,
        5 Name char(20),
        5 Number char(3);

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

  open file(Direc);

  get file(sysin) edit(Name,Number) (a(20),a(3));
  do while (¬Eof);
     write file(Direc) from(IoArea) keyfrom(Name);
     get file(sysin) edit(Name,Number) (a(20),a(3));
  end;

  close file(Direc);
end NAMELD;
```

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

Workstation VSAM keyed data sets

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

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

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

Using a SEQUENTIAL file to access a workstation VSAM keyed data set

You can open a SEQUENTIAL file that is used to access a keyed data set with either the INPUT or the UPDATE attribute.

For READ statements without the KEY option, the records are recovered in ascending key order (or in descending key order if you use the BKWD option). You can obtain the key of a record recovered in this way by using the KEYTO option.

If you use the KEY option, the record recovered by a READ statement is the one with the specified key. This READ statement positions the data set at the specified record; subsequent sequential reads recover the following records in key sequence.

WRITE statements with the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, without respect to the position of any previous access. The KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists on the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the record with the specified key; otherwise, it is the record that was accessed by the previous READ statement.

Using a DIRECT file to access a workstation VSAM keyed data set

You can open a DIRECT file that is used to access a workstation VSAM keyed data set with the INPUT, OUTPUT, or UPDATE attribute.

If you use a DIRECT OUTPUT file to add records to the data set, and if an attempt is made to insert a record with the same key as a record that already exists, the KEY condition is raised.

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

Figure 21 on page 290 shows one method you can use to update a keyed data set.

Workstation VSAM keyed data sets

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Update an ISAM keyed data set by key.                          */
/*                                                                   */
/*                                                                   */
/*  USAGE                                                            */
/*    The following commands are required to establish               */
/*    the environment variables to run this program:                 */
/*                                                                   */
/*      SET DD:DIREC=ISAM2.OUT,AMTHD(ISAM)                           */
/*      SET DD:SYSIN=ISAM3.INP,RECSIZE(80)                           */
/*                                                                   */
/*    Note: This program is using ISAM2.OUT file created by the      */
/*          previous sample program NAMELD.                          */
/*                                                                   */
/*********************************************************************/

 DIRUPDT: proc options(main);

  dcl Direc file record keyed update
      env(organization(indexed)
          recsize(23)
          keyloc(1)
          keylength(20)
         );

  dcl 1 IoArea,
        5 NewArea,
          10 Name char(20),
          10 Number char(3),
        5 Code char(1);

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

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

  on key(Direc)
  begin;
     if oncode=51 then put file(sysprint) skip edit
        ('Not found: ',Name)(a(15),a);
     if oncode=52 then put file(sysprint) skip edit
        ('Duplicate: ',Name)(a(15),a);
  end;

  open file(Direc) direct update;
```

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

```
   get file(sysin) edit (Name,Number,Code) (a(20),a(3),a(1));
   do while (¬Eof);
      put file(sysprint) skip edit (' ',Name,'#',Number,' ',Code)
          (a(1),a(20),a(1),a(3),a(1),a(1));
      select (Code);
             when('A') write file(Direc) from(NewArea) keyfrom(Name);
             when('C') rewrite file(Direc) from(NewArea) key(Name);
             when('D') delete file(Direc) key(Name);
             otherwise put file(sysprint) skip edit
                ('Invalid code: ',Name) (a(15),a);
       end;
       get file(sysin) edit (Name,Number,Code) (a(20),a(3),a(1));
   end;

   close file(Direc);
   put file(sysprint) page;

   /* Display the updated file                          */

   open file(Direc) sequential input;

   Eof = '0'b;
   on endfile(Direc) Eof = '1'b;

   read file(Direc) into(NewArea);
   do while(¬Eof);
      put file(sysprint) skip edit(Name,Number)(a,a);
      read file(Direc) into(NewArea);
   end;
   close file(Direc);
 end DIRUPDT;

An input file for this program might look like this one:

NEWMAN,M.W.         516C
GOODFELLOW,D.T.     889A
MILES,R.               D
HARVEY,C.D.W.       209A
BARTLETT,S.G.       183A
CORY,G.                D
READ,K.M.           001A
PITT,W.H.              X
ROLF,D.E.              D
ELLIOTT,D.          291C
HASTINGS,G.M.          D
BRAMLEY,O.H.        439C
```

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

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

Workstation VSAM direct data sets

The name, number, and code are read in and action taken according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

Workstation VSAM direct data sets

The statements and options allowed for workstation VSAM direct data sets are:

Table 22 (Page 1 of 4). Statements and options allowed for loading and accessing workstation VSAM direct data sets

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

Table 22 (Page 2 of 4). Statements and options allowed for loading and accessing workstation VSAM direct data sets

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

Workstation VSAM direct data sets

Table 22 (Page 3 of 4). Statements and options allowed for loading and accessing workstation VSAM direct data sets

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

Table 22 (Page 4 of 4). Statements and options allowed for loading and accessing workstation VSAM direct data sets

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

Notes:

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

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

Loading a workstation VSAM direct data set

When a direct data set is being loaded, you must open the associated file for OUTPUT. Use either a DIRECT or a SEQUENTIAL file.

For a DIRECT OUTPUT file, each record is placed in the position specified by the relative record number (or key) in the KEYFROM option of the WRITE statement (see “Using keys for workstation VSAM data sets” on page 273).

For a SEQUENTIAL OUTPUT file, use WRITE statements with or without the KEYFROM option. If you specify the KEYFROM option, the record is placed in the specified slot; if you omit it, the record is placed in the slot following the current position. There is no requirement for the records to be presented in ascending relative record number order. If you omit the KEYFROM option, you can obtain the relative record number of the written record by using the KEYTO option.

If you want to load a direct data set sequentially, without use of the KEYFROM or KEYTO options, you are not required to use the KEYED attribute.

It is an error to attempt to load a record into a position that already contains a record. If you use the KEYFROM option, the KEY condition is raised; if you omit it, the ERROR condition is raised.

Workstation VSAM direct data sets

Figure 22 on page 296 is an example of a program that defines and loads a workstation VSAM direct data set. In the PL/I program, the data set is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

If the data were in order and the keys in sequence, it would be possible to use a SEQUENTIAL file and write into the data set from the start. The records would then be placed in the next available slot and given the appropriate number. The number of the key for each record could be returned using the KEYTO option.

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

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

   dcl  Sysin file input record;
   dcl 1   In_Area,
        2   Name   char(20),
        2   Number char( 2);
   dcl Sysin_Eof bit (1) init('0'b);
   dcl Ntemp fixed(15);

      on endfile (Sysin) Sysin_Eof = '1'b;

      open file(Nos);
      read file(Sysin) into(In_Area);
      do while(¬Sysin_Eof);
         Ntemp = Number;
         write file(Nos) from(Name) keyfrom(Ntemp);
         put file(sysprint) skip edit (In_Area) (a);
         read file(Sysin) into(In_Area);
      end;

      close file(Nos);
  end CREATD;
```

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

```
This could be the input file for this program:
ACTION,G.           12
BAKER,R.            13
BRAMLEY,O.H.        28
CHEESNAME,L.        11
CORY,G.             36
ELLIOTT,D.          85
FIGGINS,E.S.        43
HARVEY,C.D.W.       25
HASTINGS,G.M.       31
KENDALL,J.G.        24
LANCASTER,W.R.      64
MILES,R.            23
NEWMAN,M.W.         40
PITT,W.H.           55
ROLF,D.E.           14
SHEERS,C.D.         21
SURCLIFFE,M.        42
TAYLOR,G.C.         47
WILTON,L.W.         44
WINSTONE,E.M.       37
```

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

Using a SEQUENTIAL file to access a workstation VSAM direct data set

You can open a SEQUENTIAL file that is used to access a direct data set with either the INPUT or the UPDATE attribute. If you use any of the options KEY, KEYTO, or KEYFROM, your file must also use the KEYED attribute.

Using READ statements

For READ statements without the KEY option, the records are recovered in ascending relative record number order. Any empty slots in the data set are skipped.

If you use the KEY option, the record recovered by a READ statement is the one with the relative record number you specify. Such a READ statement positions the data set at the specified record; subsequent sequential reads recover the following records in sequence.

Using WRITE statements

WRITE statements with or without the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, regardless of the position of any previous access. For WRITE with the KEYFROM option, the KEY condition is raised if an attempt is made to insert a record with the same relative record number as a record that already exists on the data set. If you omit the KEYFROM option, an attempt is made to write the record in the next slot, relative to the current position. The ERROR condition is raised if this slot is not empty.

You can use the KEYTO option to recover the key of a record that is added by means of a WRITE statement without the KEYFROM option.

Workstation VSAM direct data sets

Using the REWRITE or DELETE statements

REWRITE statements, with or without the KEY option, are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the record with the relative record number you specify; otherwise, it is the record that was accessed by the previous READ statement.

You can also use DELETE statements, with or without the KEY option, to delete records from the dataset.

Using a DIRECT file to access a workstation VSAM direct data set

A DIRECT file used to access a direct data set can have the OUTPUT, INPUT, or UPDATE attribute. You can read, write, rewrite, or delete records exactly as though a you were using a KEYED SEQUENTIAL file.

Figure 23 on page 299 shows a direct data set being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under workstation VSAM.

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

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

 UPDATD: proc options(main);

   dcl Nos   file record keyed
             env(organization(relative));
   dcl Sysin file input record;

   dcl Sysin_Eof bit (1) init('0'b);
   dcl   Nos_Eof bit (1) init('0'b);

   dcl 1   In_Area,
         2   Name   char(20),
         2   (CNewNo,COldNo)  char( 2),
         2   In_Area_1 char( 1),
         2   Code   char( 1);

   dcl IoField char(20);
   dcl NewNo fixed(15);
   dcl OldNo fixed(15);

   dcl oncode builtin;

   on endfile (Sysin) sysin_Eof = '1'b;
   open file (Nos) direct update;
```

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

Workstation VSAM direct data sets

```
/* trap errors */

on key(Nos)
begin;
  if oncode=51 then
     put file(sysprint) skip edit
     ('Not found:', Name) (a(15), a);
  if oncode=52 then
     put file(sysprint) skip edit
     ('Duplicate:', Name) (a(15), a);
end;

/* update the direct data set */

read file(Sysin) into(In_Area);

do while(¬Sysin_Eof);
   if CNewNo¬='  ' then
      NewNo = CNewNo;
   else
      NewNo = 0;
   if COldNo¬='  ' then
      OldNo = COldNo;
   else
      OldNo = 0;
   select(Code);
      when ('A') write file(Nos) keyfrom(NewNo) from(Name);
      when ('C')
      do;
        delete file(Nos) key(OldNo);
        write file(Nos) keyfrom(NewNo) from(Name);
      end;
      when('D') delete file(Nos) key(OldNo);
      otherwise put file(sysprint) skip list ('Invalid code:',Name);
   end;
   read file(Sysin) into(In_Area);
end;

close file(Sysin),file(Nos);

/* open and print updated file */

open file(Nos) sequential input;
on endfile (Nos) Nos_Eof = '1'b;
```

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

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

An input file for this program might look like this:

NEWMAN,M.W.         5640 C
GOODFELLOW,D.T.     89   A
MILES,R.              23 D
HARVEY,C.D.W.       29   A
BARTLETT,S.G.       13   A
CORY,G.               36 D
READ,K.M.           01   A
PITT,W.H.             55 X
ROLF,D.F.             14 D
ELLIOTT,D.          4285 C
HASTINGS,G.M.         31 D
BRAMLEY,O.H.        4928 C
```

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

Workstation VSAM direct data sets

Part 5. Using PL/I with databases

Chapter 15. Open Database Connectivity

This chapter contains information to help you use the Open Database Connectivity (ODBC) interface in your PL/I applications. With ODBC, not only can you access data from a variety of databases and file systems that support the ODBC interface, but you can do so dynamically.

Your PL/I applications that use embedded SQL for database access must be processed by a preprocessor for a particular database and have to be recompiled if the target database changes. Because ODBC is a call interface, there is no compile-time designation of the target database as there is with embedded SQL. Not only can you avoid having multiple versions of your application for multiple databases, but your application can dynamically determine which database to target.

Introducing ODBC

ODBC is a specification for an application program interface (API) that enables applications to access multiple database management systems using Structured Query Language (SQL).

ODBC permits maximum interoperability: a single application can access many different database management systems. This enables you to develop, compile, and ship an application without targeting a specific type of data source. Users can then add the database drivers, which link the application to the database management systems of their choice.

Background

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface, referred to as the *X/Open Call Level Interface*. The goal of this interface is to increase portability of applications by enabling them to become independent of any one database vendor's programming interface.

ODBC was originally developed by Microsoft for Microsoft operating systems based on a preliminary draft of X/Open CLI. Since this time, other vendors have provided ODBC drivers that run on other platforms, such as OS/2 and UNIX systems. The VA PL/I package includes the DataDirect** ODBC drivers from INTERSOLV**, Inc.

The descriptions and examples in this chapter apply to ODBC Version 3.0. For detailed information about ODBC include files, see “Using the supplied include files” on page 310.

ODBC Driver Manager

When you use the ODBC interface, your application makes calls through a Driver Manager. The Driver Manager dynamically loads the necessary driver for the database server to which the application connects. The driver, in turn, accepts the call, sends the SQL to the specified data source (database), and returns any result.

Choosing embedded SQL or ODBC

Embedded SQL and ODBC have advantages particular to them. Some of the advantages of embedded SQL are:

- Static SQL usually provides better performance than dynamic SQL. It does not have to be prepared at run time, thus reducing both processing and network traffic.
- With static SQL, database administrators have to grant users access to a package only rather than access to each table or view that is used.

Some of the advantages of ODBC are:

- It provides a consistent interface regardless of what kind of database server is used.
- You can have more than one concurrent connection.
- Applications do not have to be bound to each database on which they execute. Although VA PL/I does this bind for you automatically, it binds automatically to only one database. If you want to choose which database to connect to dynamically at run time, you must take extra steps to bind to a different database.

Using the ODBC drivers

PL/I provides an ODBC Driver Manager and a set of ODBC database drivers under an agreement with INTERSOLV, Inc.

To enable ODBC for data access in PL/I, you must install the ODBC Driver Manager and drivers by selecting the "ODBC Drivers" component during installation.

Important: During the installation process, a license file for the ODBC driver is installed on your system.

A file named `ivib.lic` is installed in *x*`:\`*plidir*`\ODBC`, where *x* and *plidir* are the drive and directory respectively, where VA PL/I is installed. (The value for *plidir* defaults to `ibmpli` for OS/2 and to `ibmpliw` for Windows.)

You must keep this file in the install directory because it is used when you run your application to verify that you are licensed to use the ODBC driver. In "Setting licensing information for ODBC Driver Manager/driver" on page 312 you learn how to use a function call to trigger the verification.

Online help

Online help is available for the ODBC drivers, both as a reference book and as context-sensitive help. The specific file names and so on may differ; you should note the names given in this section for the file names for PL/I.

Environment-specific information

The ODBC drivers are 32-bit drivers. The required network software supplied by your database system vendors must be 32-bit compliant.

OS/2 The drivers shipped for OS/2 are at the ODBC 2.1 level.

OBDC.INI is an operating system binary file located in the directory specified by the USER_INI environment variable. Since this file is binary, you cannot edit it with a text editor.

Configuring data sources

OS/2 A **data source** consists of a DBMS and any remote operating system and network necessary to access it. After the drivers have been installed, the data source must be configured using the ODBC Administrator program. Start the ODBC Administrator by double-clicking on the ODBC Administrator icon in the Tools folder.

Driver names

OS/2 The file names for the drivers that come with VA PL/I start with IB and have a file extension of .DLL. (They are dynamic link libraries.) The number in the name corresponds to the version level of the driver. When you install the ODBC drivers, your CONFIG.SYS file is modified to add the correct path to the environment variable LIBPATH.

WIN The drivers shipped for Windows are at the ODBC 3.0 level. `ODBC.INI` is a subkey of the HKEY_CURRENT_USER\\SOFTWARE\\ODBC key in the Windows NT and Windows 95 registry. The `ODBC.INI` subkey is maintained by the ODBC Administrator, which is located in the main PL/I program group. Since Windows can support multiple users, the ODBC.INI subkey is stored under unique user keys in the registry.

Configuring data sources

WIN A **data source** consists of a DBMS and any remote operating system and network necessary to access it. After the drivers have been installed, the data source must be configured using the ODBC Administrator program, which is located in the main PL/I program group. Because Windows NT and Windows 95 can host multiple users, each user must configure their own data sources. For detailed configuration information for the specific driver you wish to configure, refer to the appropriate section of the on-line help.

Driver names

WIN The file names for the drivers that come with VA PL/I start with IB and have a file extension of .DLL. (They are dynamic link libraries.) The number in the name corresponds to the version level of the driver. When you install the ODBC drivers, your Registry is modified to add the correct path to your environment.

Connecting to a data source

Your ODBC application needs to connect to the data source either using a logon dialog box or a connection string, depending on the data source.

Using a logon dialog box

Some ODBC applications display a logon dialog box when you are connecting to a data source. In these cases, the data source name has already been specified.

In the logon dialog box, do the following:

1. Type the name of the remote database or select the name of the remote database from the **Database Name** drop-down list.

   You must have cataloged any database you want to access from the client.

2. If required, type your user name (authorization ID).
3. If required, type your password.

   If you leave your user name and password blank, the ODBC application assumes you have already logged on using SQLLOGN2 (under DOS) or using User Profile Management (under OS/2). If you have not, the application returns an error. You must either type your user name and password in the dialog box or log on using SQLLOGN2 and STARTDRQ (under DOS) or using User Profile Management (under OS/2).

4. Click OK to complete the logon and to update the values in ODBC.INI.

Using a connection string

If your application requires a connection string to connect to a data source, you must specify the data source name that tells the driver which ODBC.INI section to use for the default connection information. Optionally, you may specify `attribute=value` pairs in the connection string to override the default values stored in ODBC.INI. These values are not written to ODBC.INI.

You can specify either long or short names in the connection string. The connection string has the form:

```
DSN=data_source_name[;attribute=value[;attribute=value]...]
```

An example of a connection string for INFORMIX 5 is

```
DSN=INFORMIX TABLES;DB=PAYROLL
```

Supported ODBC functions

A list of ODBC functions that are supported by the supplied drivers is found in the `read.me` file in the ODBC subdirectory of VA PL/I.

Error messages

Error messages can come from the following sources:

- An ODBC driver
- The database system
- The Driver Manager.

An error reported on an ODBC driver has the following format:

```
[vendor] [ODBC_component] message
```

ODBC_component is the component in which the error occurred. For example, an error message from INTERSOLV's SQL Server driver would look like this:

```
[INTERSOLV] [ODBC SQL Server driver] Login incorrect.
```

If you get this type of error, check the last ODBC call your application made for possible problems or contact your ODBC application vendor.

An error that occurs in the data source includes the data source name, in the following format:

```
[vendor] [ODBC_component] [data_source] message
```

With this type of message, ODBC_component is the component that received the error from the data source indicated. For example, you may get the following message from an Oracle data source:

```
[INTERSOLV] [ODBC Oracle driver] [Oracle] ORA-0919: specified length too long for CHAR column
```

If you get this type of error, you did something incorrectly with the database system. Check your database system documentation for more information or consult your database administrator. In this example, you would check your Oracle documentation.

The Driver Manager is an application that establishes connections with drivers, submits requests to drivers, and returns results to applications. An error that occurs in the Driver Manager has the following format:

```
[vendor] [ODBC DLL] message
```

vendor can be Microsoft or INTERSOLV. For example, an error from the Microsoft Driver Manager might look like this:

```
[Microsoft] [ODBC DLL] Driver does not support
this function
```

ODBC APIs from PL/I

Included with VA PL/I are ODBC include files that make it easier for you to access data bases with ODBC drivers using ODBC calls from your PL/I programs. This section describes the supplied ODBC include files, how ODBC API argument types map to PL/I data descriptions, and additional PL/I functions and considerations applicable to ODBC APIs.

For details on the ODBC APIs, see the online help.

For specific information related to an ODBC driver, such as the ODBC level or extensions supported by that driver, please refer to the specifications available with that driver.

The following illustrate how to access ODBC from PL/I programs:

ODBC APIs from PL/I

LIB Files:

OS/2 When you link your ODBC applications, you must include the import library `ODBC.LIB`, which is supplied with VA PL/I.

WIN When you link your ODBC applications, you must include the import library `ODBC32.LIB`, which is included in the ODBC SDK (from Microsoft).

CALL interface convention

Programs making ODBC calls must be compiled with the DEFAULT(BYVALUE) and LIMITS(EXTNAME(31)) compile-time options.

Using the supplied include files

The include files described and listed here are for ODBC Version 3.0.

Table 23. Supplied include files for ODBC

File name	Description
ODBCSQL.CPY	Main include for ODBC functions
ODBCEXT.CPY	Include for Microsoft's ODBC extensions
ODBCTYPE.CPY	Include for ODBC type definitions
ODBCUCOD.CPY	Include unicode
ODBCSAMP.PLI	Sample program

The supplied include files define the symbols for constant values described for ODBC APIs, mapping constants used in calls to ODBC APIs to symbols specified in ODBC guides so that argument (input and output) and function return values can be specified and tested. These files should be included in your PL/I program in order to use ODBC API calls.

In PL/I, names longer than 31 characters are truncated or abbreviated to 31 characters.

Table 24 shows the names that are longer than 31 characters, and their corresponding PL/I names.

Table 24. ODBC names truncated or abbreviated for PL/I

ODBC C #define symbol > 31 characters long	Corresponding PL/I name
SQL_AD_ADD_CONSTRAINT_DEFERRABLE	SQL_AD_ADD_CONSTR_DEFERRABLE
SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED	SQL_AD_ADD_CONSTR_INITLY_DEFERD
SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_AD_ADD_CONSTR_INITLY_IMMEDT
SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE	SQL_AD_ADD_CONSTR_NON_DEFERRABL
SQL_AD_CONSTRAINT_NAME_DEFINITION	SQL_AD_CONSTR_NAME_DEFINITION
SQL_API_ODBC3_ALL_FUNCTIONS_SIZE	SQL_API_ODBC3_ALL_FUNCTIONS_SZ
SQL_AT_CONSTRAINT_INITIALLY_DEFERRED	SQL_AT_CONSTR_INITIALLY_DEFRD
SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_AT_CONSTR_INITIALLY_IMMED
SQL_AT_CONSTRAINT_NAME_DEFINITION	SQL_AT_CONSTR_NAME_DEFINITION
SQL_AT_CONSTRAINT_NON_DEFERRABLE	SQL_AT_CONSTR_NON_DEFERRABLE
SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE	SQL_AT_DROP_TBL_CONSTR_CASCADE
SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT	SQL_AT_DROP_TBL_CONSTR_RESTRICT
SQL_CA_CONSTRAINT_INITIALLY_DEFERRED	SQL_CA_CONSTR_INITLY_DEFERRED
SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_CA_CONSTR_INITLY_IMMEDIATE
SQL_CA_CONSTRAINT_NON_DEFERRABLE	SQL_CA_CONSTR_NON_DEFERRABLE
SQL_CDO_CONSTRAINT_NAME_DEFINITION	SQL_CDO_CONSTR_NAME_DEFINITION
SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED	SQL_CDO_CONSTR_INITLY_DEFERRED
SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_CDO_CONSTR_INITLY_IMMEDIAT
SQL_CDO_CONSTRAINT_NON_DEFERRABLE	SQL_CDO_CONSTR_NON_DEFERRABLE
SQL_CT_CONSTRAINT_INITIALLY_DEFERRED	SQL_CT_CONSTR_INITLY_DEFERRED
SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_CT_CONSTR_INITLY_IMMEDIATE
SQL_CT_CONSTRAINT_NON_DEFERRABLE	SQL_CT_CONSTR_NON_DEFERRABLE
SQL_CT_CONSTRAINT_NAME_DEFINITION	SQL_CT_CONSTR_NAME_DEFINITION
SQL_DESC_DATETIME_INTERVAL_PRECISION	SQL_DESC_DATETIME_INTERVAL_PREC
SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR	SQL_DL_SQL92_INTERVAL_DAY_TO_HR
SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE	SQL_DL_SQL92_INTERVAL_DY_TO_MIN
SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND	SQL_DL_SQL92_INTERVAL_DY_TO_SEC
SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE	SQL_DL_SQL92_INTERVAL_HR_TO_MIN
SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND	SQL_DL_SQL92_INTERVAL_HR_TO_SEC
SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND	SQL_DL_SQL92_INTERVAL_MN_TO_SEC
SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH	SQL_DL_SQL92_INTERVAL_YR_TO_MTH
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	SQL_FORWARD_ONLY_CURSOR_ATTRIB1
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	SQL_FORWARD_ONLY_CURSOR_ATTRIB2
SQL_MAX_ASYNC_CONCURRENT_STATEMENTS	SQL_MAX_ASYNC_CONCURRENT_STMTS
SQL_MAXIMUM_CONCURRENT_ACTIVITIES	SQL_MAXIMUM_CONCURRENT_ACTIVITI
SQL_SQL92_FOREIGN_KEY_DELETE_RULE	SQL_SQL92_FOREIGN_KEY_DEL_RULE
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	SQL_SQL92_FOREIGN_KEY_UPD_RULE
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS	SQL_SQL92_NUMERIC_VALUE_FUNCT
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	SQL_SQL92_RELATIONAL_JOIN_OPER
SQL_TRANSACTION_ISOLATION_OPTION	SQL_TRANSACTION_ISOLATION_OPTN
SQL_TRANSACTION_READ_UNCOMMITTED	SQL_TRANSACTION_READ_UNCOMMITTD

Sample program

Mapping of ODBC C types

The data types specified in ODBC APIs are defined in terms of ODBC C types in the API definitions. The following table shows corresponding PL/I declarations for the indicated ODBC C types of the arguments.

Table 25. Mapping of ODBC C Type to PL/I Data Declarations

ODBC C type	PL/I form	Description
SQLSMALLINT	FIXED BIN(15)	Signed short integer (2 byte binary)
SQLUSMALLINT	FIXED BIN(16) UNSIGNED	Unsigned short integer (2 byte binary)
SQLINTEGER	FIXED BIN(31)	Signed long integer (4 byte binary)
SQLUINTEGER	FIXED BIN(31) UNSIGNED	Unsigned long integer (4 byte binary)
SQLREAL	FLOAT	Floating point (4 bytes)
SQLFLOAT	DOUBLE	Floating point (8 bytes)
SQLDOUBLE	DOUBLE	Floating point (8 bytes)
SQLCHAR *	CHAR(*) VARZ BYADDR	Pointer to unsigned character.
SQLHDBC	POINTER	Connection handle
SQLHENV	POINTER	Environment handle
SQLHSTMT	POINTER	Statement handle
SQLHWND	POINTER	Window handle

Setting licensing information for ODBC Driver Manager/driver

If you are using the ODBC Driver Manager/drivers shipped with VA PL/I, you need to call `ibmODBCLicInfo` immediately following a call to the `SQLConnect`, `SQLDriverConnect`, or `SQLBrowseConnect` functions. You need to pass the argument 'hdbc' to `ibmODBCLicInfo` like this:

```
sql_rc = ibmODBCLicInfo(myHDBC);
```

The `ibmODBCLicInfo` routine is included in the `ibmodlic.lib` library which must be included in the link step of your program. Refer to the sample program, `odbcsamp.pli` for more information.

Sample program using supplied include files

A sample PL/I program is supplied illustrating the use of some common ODBC functions, including:

Sample program

SQLAllocEnv
SQLAllocConnect
SQLAllocStmt
SQLBindCol
SQLBindParameter
SQLConnect
SQLDisconnect
SQLError
SQLExecDirect
SQLExecute
SQLFetch
SQLFreeConnect
SQLFreeEnv
SQLFreeStmt
SQLGetInfo
SQLNativeSQL
SQLPrepare
SQLTransact

Example Notes:

1. Use the DEFAULT(BYVALUE) and LIMITS(EXTNAME(31)) options to compile ODBC programs.
2. OS/2 For OS/2, a sample PL/I program is supplied in the `\ibmpli\samples\odbc\` directory. Use the command file `bldodbc.cmd` found in the same directory to compile and link the test program.
3. WIN For Windows, a sample PL/I program is supplied in the `\ibmpliw\samples\` directory. Use the command file `bldodbc.bat` found in the same directory to compile and link the test program.
4. The ODBC include files are available in the `\include\` subdirectory.

Chapter 16. Using DCLGEN (OS/2 only)

OS/2 VisualAge PL/I comes with a declarations generator (DCLGEN) that produces DECLARE statements you can use in your PL/I applications.

DCLGEN generates a table declaration and puts it into a file that you can include in your program. DCLGEN gets information about the definition of the table and each column within the table from the database catalog. With this information, DCLGEN produces a complete SQL DECLARE statement for the table (or view) and a matching PL/I structure declaration.

To use the declarations in your program, use the SQL INCLUDE statement.

You can invoke DCLGEN by entering DCLGEN as an OS/2 command.

If you wish to invoke DCLGEN and your table names include DBCS characters, you need to use a terminal that can input and display double-byte characters.

Understanding DCLGEN terminology

The following information explains the terms used in DCGLEN dialog boxes:

Table name
The unqualified table name for which you want DCLGEN to produce SQL data declarations. Optionally, you can qualify the table name by entering the table qualifier in the **Qualifier and Editor** dialog. DCLGEN generates a two-part table name from the table name and table qualifier.

Table qualifier
The table name qualifier. If you do not specify this value, your logon ID is assumed to be the table qualifier.

File name
The file name targeted for the declarations that DCLGEN produces.

Structure name
Name of the generated data structure which can be up to 31 characters in length.

If you leave this field blank, DCLGEN generates a name that contains the table or view name with a DCL prefix. If the table or view name consists of a DBCS string, the prefix consists of DBCS characters.

Field Name Prefix
Prefix name generated for fields in the DCLGEN output. The value you choose can be up to 28 characters in length and is used as the prefix for the field name.

For example, if you choose ABCDE, the field names generated are ABCDE001, ABCDE002, and so on.

If you leave this field blank, the field names are the same as the column names in the table or view. If the name is a DBCS string, DBCS equivalents of the suffix numbers are generated.

DCLGEN support of PL/I

A table or column name in the DECLARE statement is generated as a non-delimited identifier unless the name contains special characters and is not a DBCS string.

If you are using an SQL reserved word as an identifier, you must edit the DCLGEN output in order to add the appropriate SQL delimiters.

DCLGEN Support of PL/I

Variable names and data attributes generated by DCLGEN are derived from the information contained in databases.

Table 26. Declarations generated by DCLGEN

SQL Data Type	PL/I
SMALLINT	BIN FIXED(15)
INTEGER	BIN FIXED(31)
DECIMAL(p,s) or NUMERIC(p,s)	DEC FIXED(p,s)
FLOAT	BIN FLOAT(53)
CHAR(1)	CHAR(1)
CHAR(n)	CHAR(n)
VARCHAR(n)	CHAR(n) VARYING
LONG VARCHAR	CHAR(32700) VARYING
GRAPHIC(n)	GRAPHIC(n)
VARGRAPHIC(n)	GRAPHIC(n) VARYING
LONG VARGRAPHIC	GRAPHIC(16350) VARYING
DATE	CHAR(10)
TIME	CHAR(8)
TIMESTAMP	CHAR(26)
CLOB(nnn)	SQL TYPE IS CLOB(nnn)
BLOB(nnn)	SQL TYPE IS BLOB(nnn)
DBCLOB(nnn)	SQL TYPE IS DBCLOB(nnn)

Creating a table declaration and host structure with DCLGEN

The following example creates an SQL table declaration and a corresponding host-variable structure based on the following scenario:

Name of the file is ORG.INC
Name of the table is YEH.ORG
Default name for the structure is DCLORG

You can start DCLGEN in one of two ways:

Enter 'DCLGEN' at the OS/2 prompt

Double-click on the DCLGEN icon in the PL/I folder

Selecting a database

A dialog box appears and gives you a list of available databases in the **Select a database** list box. To select a database, move your mouse pointer to the database entry and click your left mouse button once. The entryfield above the database list displays the selected database.

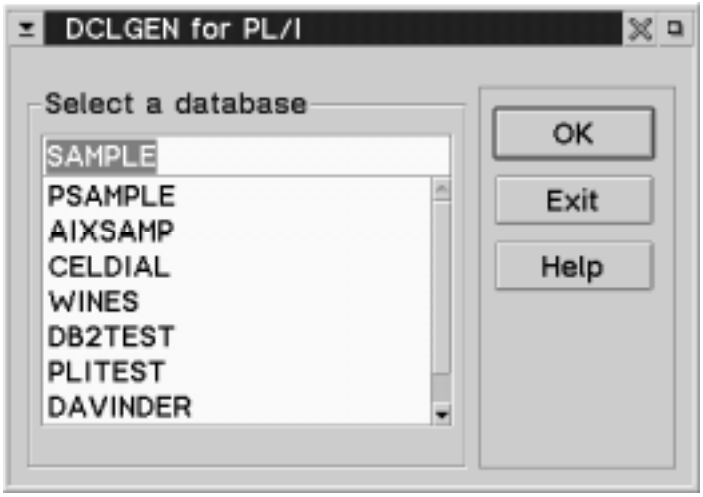


Figure 24. Selecting a database

In this example, select SAMPLE in the listbox and click on **OK** to proceed.

Logging on to your workstation

If you have not performed a local logon, a **Logon** dialog box pops up and requests that you logon.

When the **Logon** dialog box pops up, click on the **Logon** button to start a session.

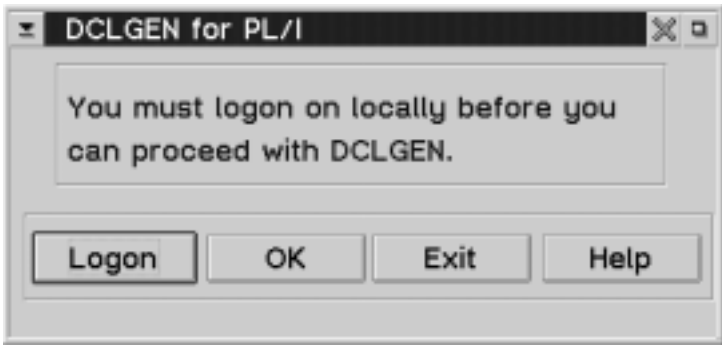


Figure 25. Logging on to a session

Creating a table declaration and host structure

After you have logged on successfully, click on **OK**.

Entering a table qualifier and editor information

After you have selected a database and logged on to your workstation, a dialog box appears requesting a table qualifier and an editor name.

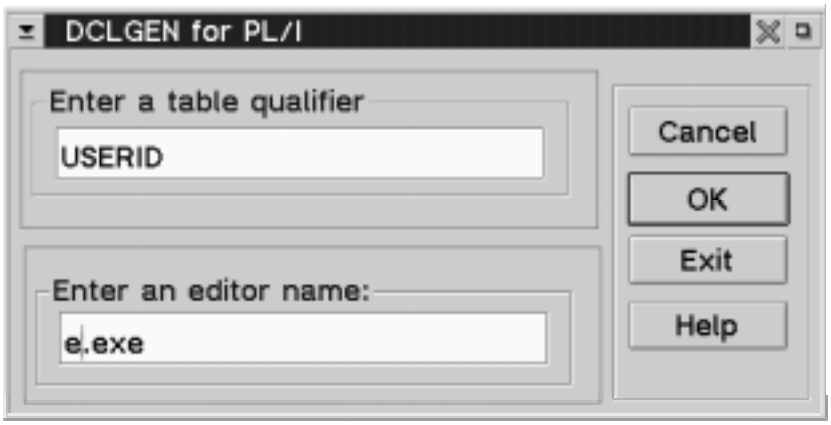


Figure 26. Requesting a table qualifier and editor

The entryfields contain default information with your local logon id as the table qualifier and the OS/2 system editor as the editor.

You can change the table qualifier or editor name (including any extensions) by typing over the default information.

Click on **OK** to proceed.

Generating a PL/I declaration

The next dialog box that appears lists all of the tables created in the database by the table qualifier.

To continue with the preceding example, select ORG in the listbox and click on **Generate**. A file named ORG.INC is created in the current directory.

Creating a table declaration and host structure

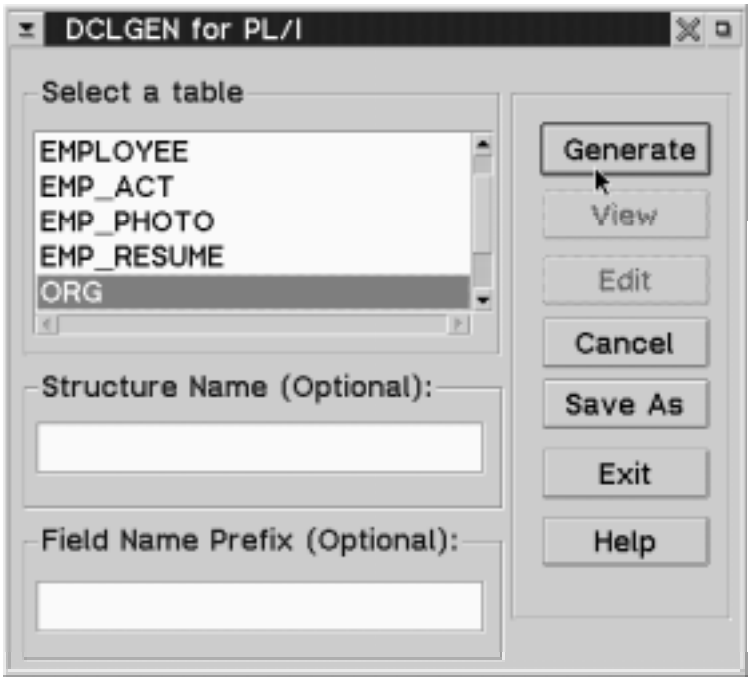


Figure 27. Selecting from the available tables in the database

DCLGEN for PL/I generates the declare statements and saves them in a file in your current directory using the table name as the file name and the .INC extension.

If you want to save the generated declaration somewhere other than the current directory, click on **Save As** before clicking on **Generate**.

Once you have changed the directory path, future generated declarations are saved in the new path until you change it.

When specifying file names, consider the following:

- If the table name contains any special characters not acceptable as part of a file name, you should specify a new file name as previously described.
- If the table name is longer than eight characters, the default name is the first eight characters and the .INC extension.

Examining the results

To browse or edit the generated declaration statement, click on **View** or **Edit**. The output file from the preceding example, ORG.INC, is shown in Figure 28.

Creating a table declaration and host structure

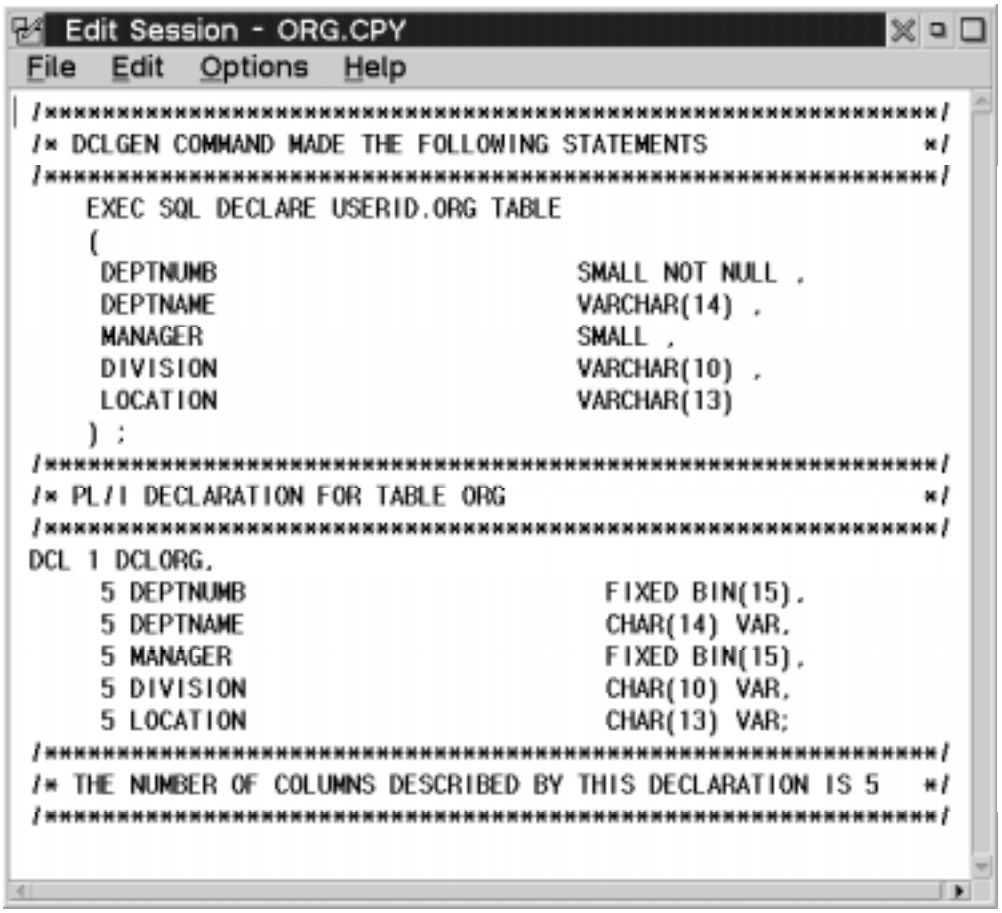


Figure 28. Viewing the declaration results

Generated statements consist of an SQL declare table statement and a PL/I structure with two levels:

Level 1
Table name prefixed by DCL

Logical Level 2
Table column names

You can specify a level 1 name in the **Structure Name** entryfield. You can also specify a field name prefix to be used in each level 2 name in the structure. For example, if you specify MYSTRUCT as the field name prefix, the level 2 names are MYSTRUCT001, MYSTRUCT002, and so on.

Exiting DCLGEN

To exit or quit DCLGEN, click on the **Exit** button from any of the panels.

Including data declarations in your program

Use the following SQL INCLUDE statement to insert the table declaration and PL/I structure declaration produced by DCLGEN into your source program:

```
exec sql
  include name ;
```

For example, to include a description for the table YEH.ORG, code:

```
exec sql
  include org ;
```

If for some reason DCLGEN produces some unexpected results, you can use the editor to tailor the output to your specific needs.

Chapter 17. Using java Dclgen (Windows only)

WIN VisualAge PL/I for Windows comes with a declarations generator (java Dclgen) that produces DECLARE statements you can use in your PL/I applications.

java Dclgen users

In order to use java Dclgen in the Windows environment, you must have the Java Developer's Toolkit (V1.1 or later) and DB2 installed on your system.

The java Dclgen tool:

- Generates a table declaration and puts it into a file that you can include in your program.
- Gets information about the definition of the table and each column within the table from the database catalog.
- Uses the information to produce a complete SQL DECLARE statement for the table (or view) and a matching PL/I structure declaration.

To use the declarations in your program, use the SQL INCLUDE statement.

If you wish to invoke java Dclgen and your table names include DBCS characters, you need to use a terminal that can input and display double-byte characters.

Understanding java Dclgen terminology

The following information explains the terms used in java Dclgen dialog boxes:

Tables
The unqualified table name for which you want java Dclgen to produce SQL data declarations. Optionally, you can qualify the table name by entering the table qualifier in the **Table Qualifier** entry field. The tool generates a two-part table name from the table name and table qualifier.

Table qualifier
The table name qualifier. If you do not specify this value, your logon ID is assumed to be the table qualifier.

Output Path for Save
The path targeted for the declarations that java Dclgen produces.

Output Filename for Save
The filename targeted for the declarations that java Dclgen produces.

Structure name
Name of the generated data structure which can be up to 31 characters in length.

If you leave this field blank, java Dclgen generates a name that contains the table or view name with a DCL prefix. If the table or view name consists of a DBCS string, the prefix consists of DBCS characters.

PL/I java Dclgen support

Field Name Prefix

Prefix name generated for fields in the javaDclgen output. The value you choose can be up to 28 characters in length and is used as the prefix for the field name.

For example, if you choose ABCDE, the field names generated are ABCDE001, ABCDE002, and so on.

If you leave this field blank, the field names are the same as the column names in the table or view. If the name is a DBCS string, DBCS equivalents of the suffix numbers are generated.

A table or column name in the DECLARE statement is generated as a non-delimited identifier unless the name contains special characters and is not a DBCS string.

If you are using an SQL reserved word as an identifier, you must edit the java Dclgen output in order to add the appropriate SQL delimiters.

PL/I java Dclgen support

Variable names and data attributes generated by java Dclgen are derived from the information contained in databases.

Table 27. Declarations generated by java Dclgen

SQL Data Type	PL/I
SMALLINT	BIN FIXED(15)
INTEGER	BIN FIXED(31)
DECIMAL(p,s) or NUMERIC(p,s)	DEC FIXED(p,s)
FLOAT	BIN FLOAT(53)
CHAR(1)	CHAR(1)
CHAR(n)	CHAR(n)
VARCHAR(n)	CHAR(n) VARYING
LONG VARCHAR	CHAR(32700) VARYING
GRAPHIC(n)	GRAPHIC(n)
VARGRAPHIC(n)	GRAPHIC(n) VARYING
LONG VARGRAPHIC	GRAPHIC(16350) VARYING
DATE	CHAR(10)
TIME	CHAR(8)
TIMESTAMP	CHAR(26)
CLOB(nnn)	SQL TYPE IS CLOB(nnn)
BLOB(nnn)	SQL TYPE IS BLOB(nnn)
DBCLOB(nnn)	SQL TYPE IS DBCLOB(nnn)

Creating a table declaration and host structure

You can start java Dclgen in one of two ways:

1. Enter 'java javaDclgen' at the MS/DOS prompt.
2. Double-click on the java Dclgen icon in the main PL/I program group.

Selecting a database

A window appears and gives you a list of available databases in the **Databases** list box. To select a database, move your mouse pointer to the database entry and click your left mouse button once. This should highlight your selection.

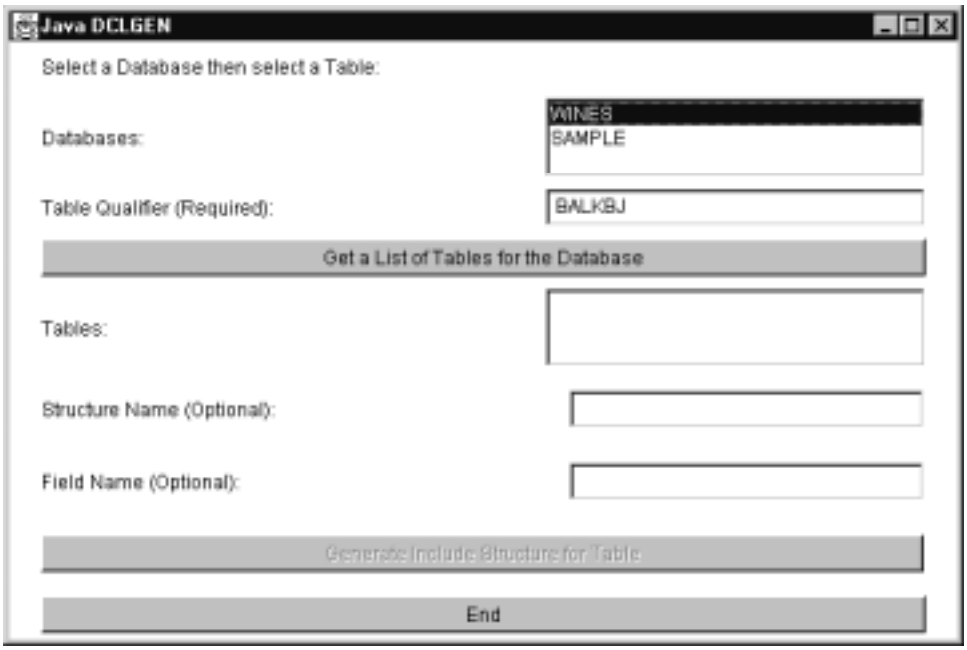


Figure 29. Selecting a database

Just below the **Databases** list box is the **Table Qualifier (Required)** entry field. This field is filled in with the current user's ID. You can use this default table qualifier or you can replace it with another valid table qualifier.

To continue, click on the **Get a List of Tables for the Database** button.

Selecting a table and generation a PL/I declaration

The **Tables** list box should be populated with the tables created in the database by the table qualifier. You can select a table in the database by clicking on it with your mouse pointer.

Creating a table declaration and host structure

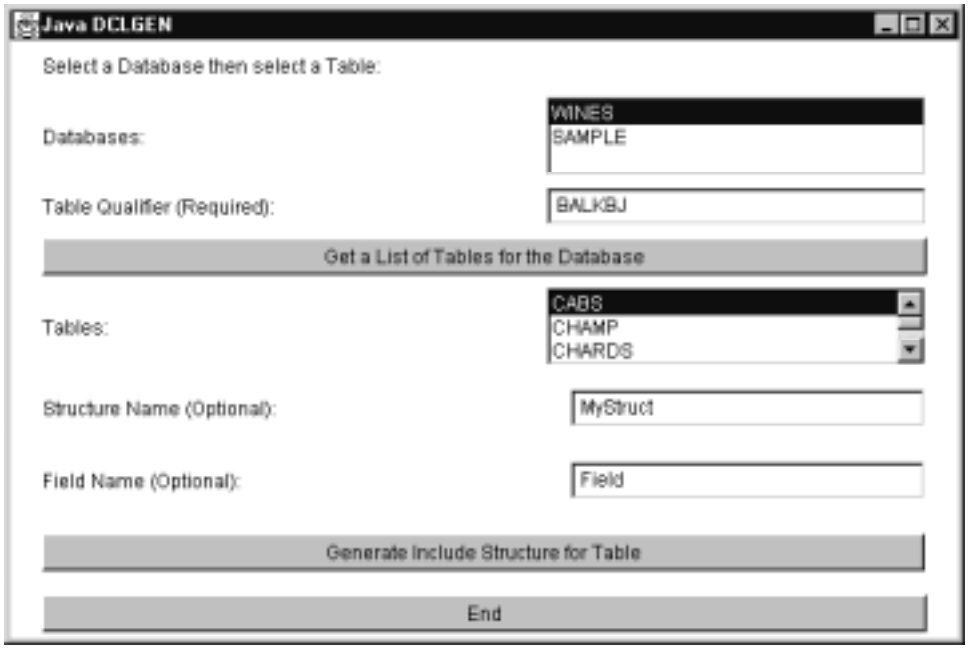


Figure 30. Display of tables created by the qualifier

You can also choose to specify a level 1 name in the **Structure Name** field as well as a field name prefix to be used in each level 2 name in the structure. For example, if you specify MYSTRUCT as the field name prefix, the level 2 names are MYSTRUCT001, MYSTRUCT002, and so on.

Click on the **Generate Include Structure for Table** button to continue.

Modifying and saving the generated PL/I declaration

The next window you should see has a text area containing the generated PL/I declaration. You can edit the contents of this area directly if needed.

Creating a table declaration and host structure

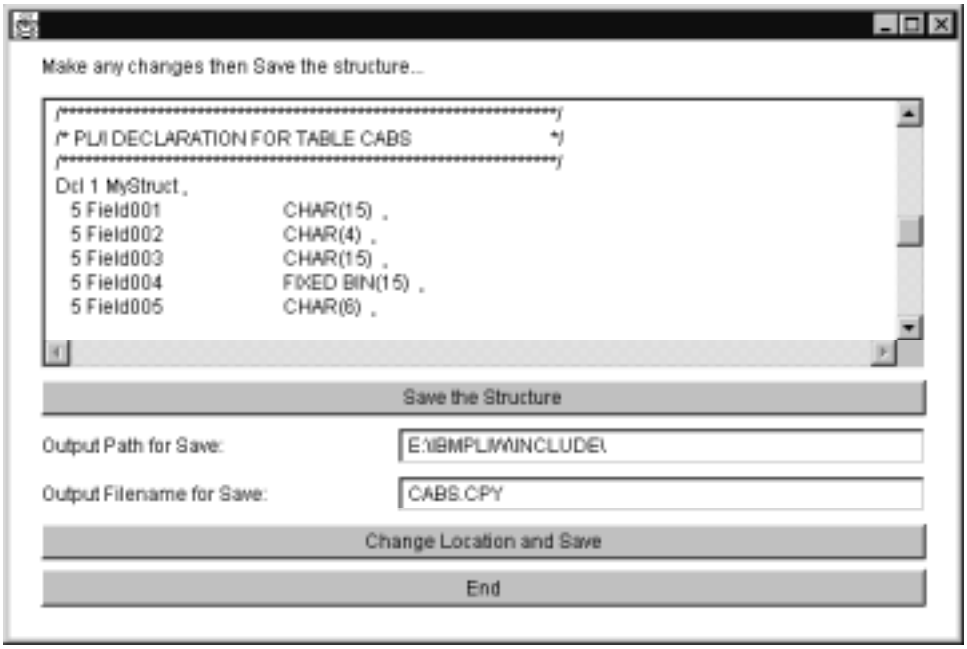


Figure 31. Generated PL/I declarations

For PL/I, java Dclgen saves the contents of the text area in a file in the `ibmpliw\include` using the table name as the filename with an extension of `.CPY`.

If you decide to save the generated declaration somewhere other than this directory, click on the **Change Location and Save** button. You can change the output directory and filename using the **Save As...** dialog.

You can change the table qualifier or editor name (including any extensions) by typing over the default information.

Note: If the table name contains any special characters that are not part of a filename, you should specify a new filename.

Exiting java Dclgen

To exit or quit java Dclgen, click on the **End** buttons successively until the application ends.

Including data declarations in your program

Use the following SQL INCLUDE statement to insert the table declaration and PL/I structure declaration produced by java Dclgen into your source program:

```
exec sql
  include name ;
```

Creating a table declaration and host structure

For example, to include a description for the table BALKBJ.ORG, code:

```
exec sql
  include org ;
```

If for some reason java Dclgen produces some unexpected results, you can use the editor to tailor the output to your specific needs.

Part 6. Advanced topics

Chapter 18. Using the Program Maintenance Utility, NMAKE

Why use NMAKE?

The Program Maintenance Utility (NMAKE) automates the process of updating project files. NMAKE compares the modification dates for one set of files (the target files) with those of another set of files (the dependent files). If any dependent files have changed more recently than the target files, NMAKE executes a series of commands to bring the targets up-to-date.

Why use NMAKE?

The most common use of NMAKE is to automate the process of updating a project after you make a change to a source file. Large projects tend to have many source files. Often, only a few of your source files need to be compiled when you make a change. You set up a special text file, called a *description* file (or *makefile*), that tells NMAKE:

- Which files depend on others
- Which commands, such as compile and link commands, need to be carried out to bring your program up-to-date.

This use of NMAKE is only one example of its power. By building suitable description files, you can use NMAKE to:

- Make backups
- Configure data files
- Run programs when data files are modified.

Running NMAKE

Run NMAKE by typing `nmake` on the operating-system command line. Supply input to NMAKE by either of two methods:

- Enter the input directly on the command line.
- Put your input into a *command file* (a text file, also called a *response file*) and enter the file name on the command line.

Press **Ctrl+C** at any time during an NMAKE run to return to the operating system.

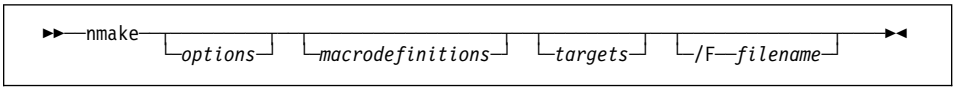
OS/2 Do not use the ampersand character (&) to combine the NMAKE command with the `cd`, `chdir`, or `set` command.

Using the command line

When using NMAKE at the command line, keep the following in mind:

- All fields are optional.
- NMAKE always looks first in the current directory for a description file called `makefile`. If `makefile` does not exist, NMAKE uses the *filename* given with the /F (specify description file) option (see “Specify description file (/F)” on page 336).

Command-line syntax



options
Specifies options that modify NMAKE's actions.

macrodefinitions
Lists macro definitions for NMAKE to use. Macro definitions that contain spaces must be enclosed by double quotation marks.

targets
Specifies the names of one or more target files to build. If you do not list any targets, NMAKE builds the first target in the description file.

***/F** filename*
Gives the name of the description file where you specify file dependencies and which commands to execute when a file is out-of-date.

The following example:

```
nmake /s "program = flash" sort.exe search.exe
```

- Invokes NMAKE with the `/s` option
- Defines a macro, assigning the string "flash" to the macro "program"
- Specifies two targets: `sort.exe` and `search.exe`

By default, NMAKE uses the file named `makefile` as the description file.

Command-line help

To display NMAKE help, type `nmake /?` at the prompt. The appropriate copyright statement appears, along with the following:

```
Usage:
      NMAKE @commandfile
      NMAKE /help

      NMAKE [/nologo] [/acdeinpqrst?] [/f makefile] [/x stderrfile]

      [macrodefs][targets]
```

Running NMAKE

```
What the options stand for
  /a       Force all targets to be built
  /c       Cryptic mode; suppress sign-on banner & warning messages
  /d       Display modification dates
  /e       Environment variables override macros in the makefile
  /i       Ignore exit codes of commands invoked
  /n       No execute mode; display commands only
  /p       Print macro definitions & target descriptions
  /q       Query if target is up to date; for use in batch files
  /r       Inference Rules from 'TOOLS.INI' to be ignored
  /s       Silent execution of commands
  /t       Touch targets with current date & time
  /?       Help message
  /help    Help message
  /nologo  Do not display sign-on banner
```

Using NMAKE command files

A command file is a *response file* used to extend command-line input to NMAKE.

You can split input to NMAKE between the command line and a command file. Use the name of a command file (preceded by @) where you normally type the input information on the command line.

Why use a command file?

Use a command file for:

- Complex and long commands you type frequently
- Strings of command-line arguments, such as macro definitions, that exceed the limit for command-line length.

A command file is not the same as a description file. For information about description files, see “Description files” on page 337.

Command file syntax

To provide input to NMAKE with a command file, type

`nmake @`*`commandfile`*

In the *commandfile* field, enter the name of a file containing the same information as is normally entered on the command line.

NMAKE treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines if you end each line except the last with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

Example

The following is a command file called `update`:

```
/s "program \
= flash" sort.exe search.exe
```

You can use this command file by typing the following command:

```
nmake @update
```

This runs NMAKE using:

- The /s option
- The macro definition "program = flash"
- The targets specified as `sort.exe` and `search.exe`
- The description file `makefile` by default

The backslash allows the macro definition to span two lines.

NMAKE options

You can use several options with NMAKE. Keep the following in mind when using options:

- Option characters are not case sensitive; /I and /i are equivalent.
- You can use either a slash or dash before the option characters; -a and /a are equivalent.

Produce error file (/X)

Syntax:`/X stderrfile`

This option produces a standard error file.

Build all targets (/A)

Syntax:`/A`

This option builds all specified targets even if they are not out-of-date with respect to their dependent files.

See “Description files” on page 337.

Suppress messages (/C)

Syntax:`/C`

This option suppresses display of the NMAKE sign-on banner, nonfatal error messages, and warning messages. To suppress the sign-on banner without suppressing other messages, use the /NOLOGO option.

Display modification dates (/D)

Syntax: `/D`

This option displays the modification date of each file when the dates of target and dependent files are checked.

See “Description files” on page 337.

NMAKE options

Override environment variables (/E)

Syntax:`/E`

This option disables inherited macro redefinition.

NMAKE *inherits* all current environment variables as macros, which can be redefined in a description file. The /E option disables any redefinition — the inherited macro always has the value of the environment variable.

Specify description file (/F)

Syntax:`/F` *`filename`*

This option specifies *filename* as the name of the description file to use. If a dash (-) is entered instead of a file name, NMAKE reads a description file from the standard input device, typically the keyboard.

If a filename is not specified, it defaults to `makefile`.

Display help (/HELP or /?)

Syntax: `/HELP` or `/?`

This option displays a brief summary of NMAKE syntax.

Ignore exit codes (/I)

Syntax:`/I`

This option ignores exit codes (also called error level or return codes) returned by programs such as compilers or linkers called by NMAKE. If this option is not specified, NMAKE ends when any program returns a nonzero exit code.

Display commands (/N)

Syntax:`/N`

This option causes NMAKE commands to be displayed but not executed. Use the /N option to:

- Check which targets are out-of-date with respect to their dependents
- Debug description files

Suppress sign-on banner (/NOLOGO)

Syntax:`/NOLOGO`

This option suppresses the sign-on banner display when NMAKE is started. If you want to suppress nonfatal error messages and warnings as well, use the suppress messages (/C) option.

Print macro and target definitions (/P)

Syntax: /P

This option writes out all macro definitions and target definitions. Output is sent to the standard output device (typically the display).

Return exit code (/Q)

Syntax: /Q

This option causes NMAKE to return either of the following:

- An exit code of zero if all targets built during an NMAKE run are up-to-date
- An exit code other than zero if they are not up-to-date

Use this option to run NMAKE from within a batch file.

Ignore TOOLS.INI file (/R)

Syntax: /R

This option ignores the following:

- All inference rules and macros contained in the TOOLS.INI file
- All predefined inference rules and macros

Suppress command display (/S)

Syntax: /S

This option suppresses the display of commands as they are executed by NMAKE. It does not suppress the display of messages generated by the commands themselves.

The /N command (Display Commands) takes precedence over the /S option. If you use /N and /S together, commands are displayed but not executed.

Change target modification dates (/T)

Syntax: /T

This option changes or “touches” the modification dates for out-of-date target files to the current date. No commands are executed, and the target file is left unchanged.

Description files

NMAKE uses a description file to determine what to do. In its simplest form, a description file tells NMAKE which files depend on others and which commands need to be executed if a file changes.

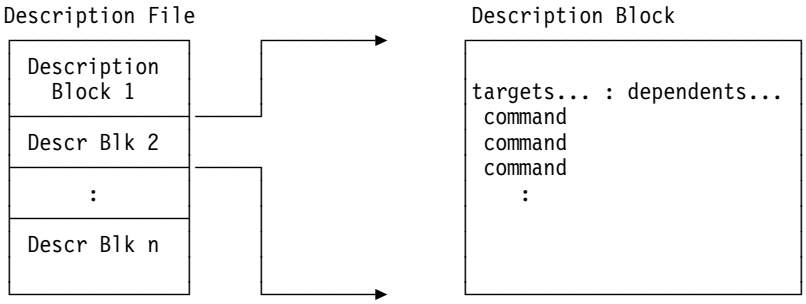
A description file looks like this:

Description files

```
targets...: dependents...
     command
        :

targets... : dependents...
     command
```

Description blocks

A dependent relationship between files is defined in a *description block*. A description block indicates the relationship among various parts of the program. It contains commands to bring all components up to date. The description file can contain up to 1048 description blocks.



Special features

The following are special features of description files and blocks:

- Description files can contain macro definitions and use macros in description blocks. Macros allow easy substitution of one text string for another.
- Description files can contain inference rules. Inference rules allow NMAKE to infer which commands to execute based on the filename extensions used for targets and dependents.
- You can specify directories for NMAKE to search for dependent files by using the following syntax:

  ```
  targets : {directory1;directory2...}dependents
  ```

  NMAKE searches the current directory first, then *directory1*, *directory2*, and so on.
- A command can be placed on the same line as the target and dependent files by using a semicolon (;) as depicted below:

  ```
  targets... : dependents... ; command
  ```
- A long command can span several lines if each line ends with a backslash (\ ):

  ```
  command \
    continuation of command
  ```
- The execution of a command can be modified if you precede the command with special characters.

- If you do not specify a command in a description block, NMAKE looks for an inference rule to build the target.
- Wild card characters (* and ?) can be used in description blocks. For example, the following description block compiles all source files with the .PLI extension:

```
astro.exe : *.pli
  pli $**
```

- NMAKE expands the *.pli specification into the complete list of PL/I files in the current directory. $** is a complete list of dependents specified for the current target.
- NMAKE uses several punctuation characters in its syntax. To use one of these characters as a literal character, place an escape character (^) in front of it. For a list of punctuation characters, see “Escape characters” on page 354.
- Normally a target file can appear in only one description block. A special syntax allows you to use a target in several description blocks.
- A special syntax allows you to determine the drive, path, base name, and extension of the first dependent file in a description block.

Targets in several description blocks

Using a file as a target in more than one description block causes NMAKE to end. You can overcome this limitation by using two colons (::) as the target/dependent separator instead of one colon.

The following description block is permissible:

```
X :: A
  command
X :: B
  command
```

The following causes NMAKE to end:

```
X : A
  command
X : B
  command
```

It is permissible to use single colons if the target/dependent lines are grouped above the same commands. The following is permissible:

```
X : A
X : B
  command
```

Double colon (::) target/dependent separator example

```
target.lib :: a.asm b.asm c.asm
  ml a.asm b.asm c.asm
  ilib target a.obj b.obj c.obj

target.lib :: d.pli e.pli
  pli d.pli
  pli e.pli
  ilib target d.obj e.obj
```

These two description blocks both update the library named `target.lib`. If any of the assembly-language files have changed more recently than the library file, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the PL/I language files have changed, NMAKE executes the second group of commands to compile the PL/I files and update the library.

Using macros

Macros provide a convenient way to replace one string with another in the description file. The text is automatically replaced each time NMAKE is run. This feature makes it easy to change text throughout the description file without having to edit every line that uses the text.

Two common uses of macros are:

- To create a standard description file for several projects. The macro represents the file names in commands. These file names are defined when you run NMAKE. When you switch to a different project, changing the macro changes the file names NMAKE uses throughout the description file.
- To control the options that NMAKE passes to the compiler, assembler, or linker. When using a macro to specify the options, you can quickly change the options throughout the description file in one easy step.

A macro can be defined:

In a description file
On the command line
In `TOOLS.INI`
Through inheritance from environment variables

Macros example

```
program = flash
c = ilink
options =

$(program).exe : $(program).obj
  $c $(options) $(program).obj;
```

The example above defines three macros. The description block executes the following commands:

```
flash.exe : flash.obj
  ilink  flash.obj;
```

Special features

Macros have the following special features:

- When using a macro, you can substitute text in the macro itself.
- Several macros have been predefined for special purposes.
- If a macro is defined more than once, precedence rules govern which definition is used.
- You can also put macros into your `TOOLS.INI` file.

Macros in a description file

Before using a macro, you need to define it, either on the NMAKE command line or in your description file. Description file macro definitions look like this:

```
macroname = macrostring
```

Macro names can be any combination of alphanumeric characters and the underscore character (_), and they are case-sensitive. A macro string can be any string of characters.

The first character of the macro name must be the first character on the line. NMAKE ignores any spaces before or after the equal sign (=).

The macro string can be a null string and can contain embedded spaces. Do not enclose the macro string in quotation marks in the description file; quotation marks are used only when you define macros on the command line.

Macros on the command line

Before using a macro, you need to define it, either on the NMAKE command line or in your description file. Command-line macro definitions look like this:

```
macroname=macrostring
```

Using macros

No spaces can surround the equal sign. If you embed spaces, NMAKE might misinterpret your macro. If your macro string contains embedded spaces, enclose it in double quotation marks (") like this:

```
macroname="macro string"
```

You can also enclose the entire macro definition in double quotation marks (") like this:

```
"macroname = macro string"
```

Macro names can be any combination of alphanumeric characters and the underscore character (_), and they are case-sensitive. A macro string can be any string of characters or a null string.

Inherited macros

NMAKE *inherits* all current environment variables as macros. For example, if you have a PATH environment variable defined as `PATH = C:\TOOLS\BIN`, the string `C:\TOOLS\BIN` is substituted when you use PATH in the description file.

You can redefine inherited macros by including a line such as the example above in a description file. While NMAKE is executing, the macro takes on the redefined definition. When NMAKE terminates, however, the environment variable resumes its original value.

The Override Environment Variables (/E) option disables inherited macro redefinition. If you use this option, NMAKE ignores any attempt to redefine an inherited macro.

The macro name, for any macros that you define, is case sensitive. For example, consider this macro:

```
UPPER=UpperCase
```

In this example, $(UPPER) returns the value, but $(upper) does not. Inherited macro names (i.e. those created automatically from environment variables) must always be UPPERCASE.

Defined macros

After you have defined a macro, you can use it anywhere in your description file with the following syntax:

```
$(macroname)
```

The parentheses are not required if the macro name is only one character long. To use a dollar sign ($) without using a macro, enter two dollar signs ($$), or use the caret (^) before the dollar sign as an escape character.

When NMAKE runs, it replaces all occurrences of $(macroname) with the defined macro string. If the macro is undefined, nothing is substituted. After a macro is defined, you can cancel it only with the !UNDEF directive.

Macro substitutions

Just as you use macros to substitute text within a description file, you use the following syntax to substitute text within a macro:

```
$(macroname: string1 = string2)
```

Every occurrence of *string1* is replaced by *string2* in *macroname*. Spaces between the colon and *string1* are considered part of *string1*. If *string2* is a null string, all occurrences of *string1* are deleted from the macro. The colon (:) must immediately follow *macroname*.

The replacement of *string1* with *string2* in the macro is not a permanent change. If you use the macro again without a substitution, you get the original unchanged macro.

Example

```
SOURCES = one.pli two.pli three.pli
program.exe : $(SOURCES:.pli=.obj)
  ilink $**;
```

The example above defines a macro called SOURCES, which contains the names of three PL/I source files. With this macro, the target/dependent line substitutes the `.obj` extension for the `.pli` extension. Thus, NMAKE executes the following command:

```
ilink one.obj two.obj three.obj;
```

$** is a special macro that translates to all dependent files for a given target.

Special macros

NMAKE predefines several macros. The first six macros below return one or more file specifications for the files in the target/dependent line of a description block. Except where noted, the file specification includes the path of the file, the base filename, and the filename extension.

Macro	Value
$@	The specification of the target file.
$*	The base name (without extension) of the target file. Path information is also returned if the path was specified as part of the target filename. This macro cannot be used in a dependent list.
$**	The specifications of the dependent files.
$?	The specifications for only those dependent files that are out-of-date with respect to the targets.
$<	The specification of a single dependent file that is out-of-date with respect to the targets. This macro is used only in inference rules.
$$@	The file specification of the target that NMAKE is currently evaluating. This is a dynamic dependency parameter, used only in dependent lists.

Special macros

$(AS) The string MASM, which is the command to run the Macro Assembler (MASM). You can redefine this macro to use a different command.

$(MAKE) The command name used to run NMAKE. This macro is used to invoke NMAKE recursively. If you redefine this macro, NMAKE issues a warning message.

NMAKE executes the command line in which $(MAKE) appears, even if the display commands (/N) option is on.

$(MAKEFLAGS) The NMAKE options currently in effect. You cannot redefine this macro.

The special macros $** and $$@ are the only exceptions to the rule that macro names longer than one character must be enclosed in parentheses.

You can append characters to any of the first six macros in this list to modify the meaning of the macro. However, you cannot use macro substitutions in these macros.

Special macros examples

```
trig.lib : sin.obj cos.obj arctan.obj
  !ilib trig.lib $?
```

In this example, the macro $? represents the names of all dependent files that are out-of-date with respect to the target file. The exclamation point (!) preceding the ILIB command causes NMAKE to execute the ILIB command once for each dependent file in the list. As a result of this description, the ILIB command causes NMAKE to execute the ILIB command once for each dependent file in the list. As a result of this description, the ILIB command is executed up to three times, each time replacing a module with a newer version.

```
DIR=c:\include
$(DIR)\globals.inc : globals.inc
 copy globals.inc $@
$(DIR)\types.inc : types.inc
 copy types.inc $@
$(DIR)\macros.inc : macros.inc
 copy macros.inc $@
```

This example shows how to update a group of include files. Each of the files, `globals.inc`, `types.inc`, and `macros.inc`, in the directory `c:\include` depends on its counterpart in the current directory. If one of the include files is out-of-date, NMAKE replaces it with the file of the same name from the current directory.

The following description file, which uses the special macro $$@, is equivalent:

```
DIR=c:\include
$(DIR)\globals.inc $(DIR)\types.inc $(DIR)\macros.inc : $$(@F)
!copy $? $@
```

The special macro $$(@F) signifies the file name (without the path) of the current target.

When NMAKE evaluates the description block, it evaluates the three targets, one at a time, with respect to their dependents. Thus, NMAKE first checks whether `c:\include\globals.inc` is out-of-date compared with `globals.inc` in the current directory. If so, it executes the command to copy the dependent file `globals.inc` to the target. NMAKE repeats the procedure for the other two targets.

Note that on the command line, the macro $? refers to the dependent for this target. The macro $@ specifies the full file specification of the target file.

File-specification parts

A full file specification gives the base name of the file, the file-name extension, and the path. The path provides the disk-drive identifier and the sequence of directories needed to locate the file on the disk.

For example, the file specification

```
c:\source\prog\sort.obj
```

has the following parts:

```
Path Name            c:\source\prog
Base File Name       sort
File-Name Extension  .obj
```

Characters that modify special macros

The following six macros all resolve to a file specification (or possibly several file specifications for $** and $?):

```
$*     $@     $**     $<     $?     $$@
```

You can append characters to any of these macros to modify the file name returned by the macro. Depending on which character you use, parts of the full file specification are returned:

	Appended Character			
File Part Returned	D	F	B	R
File Path	Yes	No	No	Yes
Base File Name	No	Yes	Yes	Yes
File Name Extension	No	Yes	No	No

Modified special macros example

If the macro $@ has the value

```
c:\source\prog\sort.obj
```

then the following values are returned for the modified macro:

Inference rules

Macro	Value
$(@D)	c:\source\prog
$(@F)	sort.obj
$(@B)	sort
$(@R)	c:\source\prog\sort

Modified macros are always longer than a single character — they must be enclosed by parentheses when used.

Macro precedence rules

When the same macro is defined in more than one place, the definition with the highest priority is used:

Priority	Definition
1 (Highest)	Command line
2	Description file
3	Environment variables
4	TOOLS.INI file
5 (Lowest)	Predefined macros (such as CC and AS)

If you invoke NMAKE with the Overriding Macro Definitions (/E) option, macros defined by environment variables take precedence over those defined in a description file.

Inference rules

Inference rules are templates from which NMAKE infers what to do with a description block when no commands are given. Only those extensions defined in a .SUFFIXES list can have inference rules. The extensions .c, .obj, .asm, and .exe are automatically included in .SUFFIXES.

PL/I programmers

You must add PL/I file extensions manually using the .SUFFIXES pseudotarget. See ".SUFFIXES Pseudotarget" on page 352.

When NMAKE encounters a description block with no commands, it looks for an inference rule that specifies how to create the target from the dependent files, given the two file extensions. Similarly, if a dependent file does not exist, NMAKE looks for an inference rule that specifies how to create the dependent from another file with the same base name.

NMAKE applies an inference rule only if the base name of the file it is trying to create matches the base name of a file that already exists.

In effect, inference rules are useful only when there is a one-to-one correspondence between the files with the "from" extension and the files with the "to" extension. You

cannot, for example, define an inference rule that inserts a number of modules into a library.

The use of inference rules eliminates the need to put the same commands in several description blocks. For example, you can use inference rules to specify a single `pli` command that changes any PL/I source file (with a `.pli` extension) to an object file (with a `.obj` extension).

You define an inference rule by including text of the following form in your description file or in your `TOOLS.INI` file — see “Special Features”.

```
.fromext.toext:
commands
:
```

The elements of the inference rule are:

fromext	The file-name extension for dependent files to build a target
toext	The file-name extension for target files to be built
commands	The commands to build the *toext* target from the *fromext* dependent.

For example, an inference rule to convert PL/I source files (with the `.pli` extension) to PL/I object files (with the `.obj` extension) is

```
.pli.obj:
 pli $<
```

The special macro $< represents the name of a dependent out-of-date relative to the target.

Special features

- You can specify a path where NMAKE should look for target and dependent files used in inference rules.
- Inference rules are predefined for compiling and linking C programs, and for assembling programs.
- NMAKE looks for inference rules in the `TOOLS.INI` file if it cannot find a rule in a description file.
- Only those extensions defined in a .SUFFIXES list can have inference rules. The extensions `.c`, `.obj`, `.asm`, and `.exe` are automatically included in .SUFFIXES.
- You must add PL/I file extensions manually using the .SUFFIXES pseudotarget. See “.SUFFIXES Pseudotarget” on page 352.

Inference rules

Inference rules example

```
.obj.exe:
  ilink $<;

example1.exe: example1.obj

example2.exe: example2.obj
  ilink /co example2,,,libv3.lib
```

The first line above defines an inference rule that causes the ILINK command to create an executable file whenever a change is made in the corresponding object file. The file name in the inference rule is specified with the special macro $< so that the rule applies to any `.obj` file with an out-of-date executable file.

When NMAKE does not find any commands in the first description block, it checks for a rule that might apply and finds the rule defined on the first two lines of the description file. NMAKE applies the rule, replacing $< with `example1.obj` when it executes the command, so that the ILINK command becomes

```
ilink example1.obj;
```

NMAKE does not search for an inference rule when examining the second description block, because a command is explicitly given.

Inference-rule path specifications

When defining an inference rule, you can indicate to NMAKE where to look for target and dependent files. Use the following syntax:

```
{frompath}.fromext{topath}.toext
 commands
 :
```

NMAKE looks in the directory specified by *frompath* for files with the *fromext* extension. It executes the commands to build files with the *toext* extension in the directory specified by *topath*.

Predefined inference rules

NMAKE predefines several inference rules:

Table 28. NMAKE Predefined Inference Rules

Inference Rule	Command Action	Default
.c.obj	$(CC) $(CFLAGS) /c $*.c	icc /c $*.c
.c.exe	$(CC) $(CFLAGS) $*.c	icc $*.c
.asm.obj	$(AS) $(AFLAGS) $*;	masm $*;

- The first two rules automatically compile and link C programs.
- The last rule automatically assembles programs.
- The above are the most often used predefined inference rules. For a complete list of predefined inference rules, execute a makefile and specify the /p option. All available inference rules will be displayed.

Directives

Using directives, you can construct description files similar to batch files. NMAKE provides directives that:

- Conditionally execute commands
- Display error messages
- Include the contents of other files
- Turn some NMAKE options on or off

Each directive begins with an exclamation point (!) in the first column of the description file. Spaces can be placed between the exclamation point and the directive keyword.

The list below describes the directives:

!IF *expression* Executes the statements between the !IF keyword and the next !ELSE or !ENDIF directive if *expression* evaluates to a nonzero value.

The *expression* used with the !IF directive can consist of integer constants, string constants, or exit codes returned by programs. Integer constants can use the C unary operators for numerical negation (-), one's complement (~), and logical negation (!). You can also use any of the C binary operators listed below:

Operator	Description
+	Addition
-	Subtraction
*****	Multiplication

Directives

/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

- You can use parentheses to group expressions.
- Values are assumed to be decimal values unless specified with a leading 0 (octal) or leading 0x (hexadecimal).
- Strings are enclosed by quotation marks ("). You can use the equality (==) and inequality (!=) operators to compare two strings.
- You can invoke a program in an expression by enclosing the program name in square brackets ([ ]). The exit code returned by the program is used in the expression.

!ELSE Executes the statements between the !ELSE and !ENDIF directives if the statements preceding the !ELSE directive were not executed.

!ENDIF Marks the end of the !IF, !IFDEF, or !IFNDEF block of statements.

!IFDEF *macroname*
Executes the statements between the !IFDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is defined in the description file. If a macro has been defined as null, it is still considered to be defined.

!IFNDEF *macroname*
Executes the statements between the !IFNDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is not defined in the description file.

!UNDEF *macroname*
Undefines a previously defined macro.

!ERROR *text* Prints text and then stops execution.

!INCLUDE *filename*
Reads and evaluates the file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise, it looks only in the current directory. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.

!CMDSWITCHES {+|-}*opt*
Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If no options are specified, the options are reset to the values they had when NMAKE was started. To turn an option on, precede it with a plus sign (+); to turn it off, precede it with a minus sign (-). This directive updates the MAKEFLAGS macro.

See “Special macros” on page 343.

Directives example

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe:winner.obj
!IFDEF DEBUG
! IF "$(DEBUG)"=="y"
     ilink /de winner.obj;
! ELSE
     ilink winner.obj;
! ENDIF
!ELSE
! ERROR Macro named DEBUG is not defined.
!ENDIF
```

The directives in this example do the following:

- The !INCLUDE directive causes the file `infrules.txt` to be read and evaluated as if it were part of the description file.
- The !CMDSWITCHES directive turns on the /D option, which displays the dates of the files as they are checked.
- If `winner.exe` is out-of-date with respect to `winner.obj`, the !IFDEF directive checks to see whether the macro DEBUG is defined. If it is defined, the !IF directive checks to see whether it is set to y. If it is, the linker is invoked with the /DE option; otherwise, it is invoked without the /DE. If the DEBUG macro is not defined, the !ERROR directive prints the message and NMAKE stops executing.

Pseudotargets

A *pseudotarget* is a target in a description block that is not a file. Instead, it is a name that serves as a "handle" for building a group of files or executing a group of commands. In the following example, UPDATE is a pseudotarget:

Directives

```
UPDATE: *.*
  !copy $** a:\product
```

When NMAKE evaluates a pseudotarget, it always considers the dependents to be out-of-date. In the description above, NMAKE copies each of the dependent files to the specified drive and directory.

NMAKE predefines several pseudotargets for special purposes.

See “Predefined pseudotargets.”

Predefined pseudotargets

NMAKE predefines several pseudotargets that provide special rules within a description file:

.SILENT Pseudotarget

Syntax: .SILENT : dependents...

This pseudotarget suppresses the display of executed commands for a single description block. The /S option does the same thing for all description blocks.

See “Suppress command display (/S)” on page 337.

.IGNORE Pseudotarget

Syntax: .IGNORE : dependents...

This pseudotarget ignores exit codes returned by programs for a single description block. The /I option does the same thing for all description blocks.

See “Ignore exit codes (/I)” on page 336.

.SUFFIXES Pseudotarget

Syntax: .SUFFIXES : extensions...

This pseudotarget defines file extensions to try when NMAKE needs to build a target file for which no dependents are specified. NMAKE searches the current directory for a file with the same name as the target file and an extension in <extensions...>. If NMAKE finds such a file, and if an inference rule applies to the file, NMAKE treats the file as a dependent of the target.

The .SUFFIXES pseudotarget is predefined as

```
.SUFFIXES : .obj .exe .c .asm
```

To add extensions to the list, specify .SUFFIXES : followed by the new extensions. For example, the following would enable you to write interence rules for PL/I source files.

```
.SUFFIXES: .pli
```

To clear the list, specify

```
.SUFFIXES:
```

Only those extensions specified in .SUFFIXES can have inference rules. NMAKE ignores inference rules unless the extensions have been specified in a .SUFFIXES list.

.PRECIOUS Pseudotarget

Syntax: .PRECIOUS : targets...

This pseudotarget tells NMAKE not to delete a target even if the commands that build it are terminated or interrupted. This pseudotarget overrides the NMAKE default. By default, NMAKE deletes the target if it cannot be sure that the target was built successfully.

For example,

```
.PRECIOUS : tools.lib
tools.lib : a2z.obj z2a.obj
 command
  :
```

If the commands to build `tools.lib` are interrupted, leaving an incomplete file, NMAKE does not delete the partially built `tools.lib`.

The pseudotarget .PRECIOUS is useful only in limited circumstances. Most professional development tools have their own interrupt handlers and "clean up" when errors occur.

Inline files

You may need to issue a command in the description file with a list of arguments exceeding the command-line limit of the operating system. Just as NMAKE supports the use of command files, it can also generate inline files which are read as response files by other programs.

To generate an inline file, use the following syntax for your description block:

```
target : dependents
  command @<<[filename]
inline file text
<< [KEEP | NOKEEP]
```

All of the text between the two sets of double less than signs (<<) is placed into an inline file and given the name *filename*. You can refer to the inline file at a later time by using *filename*. If *filename* is not given, NMAKE gives the file a unique name in the directory specified by the TMP environment variable if it is defined. Otherwise, NMAKE creates a unique file name in the current directory.

The inline file can be temporary or permanent. If you do not specify otherwise, or if you specify the keyword NOKEEP, the inline file is temporary. Specify KEEP to retain the file.

Inline files

The at sign (@) is not part of the NMAKE syntax but is the typical character used by utilities to designate a file as a response file.

Inline files example

```
math.lib :  add.obj sub.obj mul.obj div.obj
  ilib @<<
math.lib
add.obj  sub.obj  mul.obj  div.obj
/L:listing
<<
```

The above example creates an inline file and uses it to invoke the Library Manager (ILIB). The inline file is used as a response file by (ILIB). It specifies which library to use, the commands to execute, and the listing file to produce. The inline file contains the following:

```
math.lib
add.obj  sub.obj  mul.obj  div.obj
/L:listing
```

Because no file name is listed after the ILIB command, the inline file is given a unique name and placed into the current directory (or the directory defined by the TMP environment variable).

Escape characters

NMAKE uses the following punctuation characters in its syntax:

```
(           )          #          $          ^          \
{           }          !          @          -
```

To use one of these characters in a command and not have it interpreted by NMAKE, use a caret (^) in front of the character.

For example,

```
BIG^#.PLI
```

is treated as

```
BIG#.PLI
```

With the caret, you can include a literal newline character in a description file. This capability is useful in macro definitions, as in the following example:

```
XYZ=abc^<ENTER>
def
```

The effect is equivalent to the effect of assigning the C-style string abc\ndef to the XYZ macro. Note that this effect differs from the effect of using the backslash (\ ) to continue a line. A newline character that follows a backslash is replaced with a space.

NMAKE ignores a caret that is not followed by any of the characters it uses in its syntax. A caret that appears within quotation marks is not treated as an escape character.

The escape character cannot be used in the command portion of a dependency block.

Characters that modify commands

Any of three characters can be placed in front of a command to modify how the command is run:

— (dash) Turns off error checking for the command

@ (at sign) Suppresses display of the command

! (exclamation point)
Executes the command for each dependent file

Spaces can separate the modifying character from the command. Any command on a separate line — whether modified or not — must be indented by one or more spaces or tabs.

You can use more than one character to modify a single command.

Turn error checking off (-)

Syntax: -[n] command

The /I option globally turns command error-checking off. The dash (-) command modifier overrides the global setting to turn error checking off for commands individually. This modifier is used in two ways:

- A dash without a number turns off all error checking.
- A dash followed by a number causes NMAKE to abort only if the exit code returned by the command is greater than the number.

See “Ignore exit codes (/I)” on page 336.

Dash command modifier examples

```
light.lst : light.txt
  - flash light.txt
```

In the example above, NMAKE never ends, regardless of the exit code returned by `flash`.

```
light.lst : light.txt
  -1 flash light.txt
```

In the example above, NMAKE ends if the exit code returned by `flash` is greater than 1.

Characters that modify commands

Suppress command display (@)

Syntax: @ command

The /S option globally suppresses the display of commands while NMAKE is running. The at sign (@) modifier suppresses the display for individual commands.

Regardless of the /S option or the @ modifier, output generated by the command itself always appears.

See “Suppress command display (/S)” on page 337.

At sign (@) command modifier example

Suppress Command Display (@)

```
sort.exe:sort.obj
  @ echo sorting
```

The command line calling the `echo` command is not displayed. The output of the `echo` command, however, is displayed.

Execute command for dependents (!)

Syntax: ! command

The exclamation-point command modifier causes the command to be executed for each dependent file if the command uses one of the special macros $? or $**. The $? macro refers to all dependent files out-of-date with respect to the target. The $** macro refers to all dependent files in the description block.

See “Special macros” on page 343.

Exclamation point (!) command modifier examples

```
leap.txt : hop.asm skip.c jump.pli
  ! print $** lpt1:
```

The example above executes the following three commands, regardless of the modification dates of the dependent file:

```
print hop.asm lpt1:
print skip.c lpt1:
print jump.pli lpt1:

leap.txt : hop.asm skip.c jump.pli
  ! print $? lpt1:
```

The example above executes the print command only for those dependent files with modification dates later than that of the `leap.txt` file. If `hop.asm` and `jump.pli` have modification dates later than `leap.txt`, the following two commands are executed:

```
print hop.asm lpt1:
print jump.pli lpt1:
```

EXTMAKE Syntax

Description files can use a special syntax to determine the drive, path, base name, and extension of the first dependent file in a description block. This syntax is called the *extmake* syntax.

The characters `%s` represent the complete file specification of the first dependent file. Various parts of the file specification are represented using the following syntax:

`%<parts>`

<parts>
A combination of the following letters:

d Drive
p Path
f Base name
e Extension

For example, to specify the drive and path name of the first dependent file in a description block, use:

```
%<dp>
```

The percent symbol (%) is a replacement in DOS, Windows, and OS/2 command lines. To use the percent symbol in command-line arguments, use a double percent (%%).

Macros and inference rules in TOOLS.INI

You can place either macros or inference rules in your `TOOLS.INI` file. NMAKE looks for the `TOOLS.INI` file first in the current directory and then in the directory indicated by the INIT environment variable.

If NMAKE finds a `TOOLS.INI` file, it looks for the following tag:

```
[nmake]
```

You can place macros and inference rules below this tag in the same format you would use in a description file.

If a macro or inference rule is defined in both the `TOOLS.INI` file and the description file, the definition in the description file takes precedence. Also, if you use the /R option, the `TOOLS.INI` file is ignored.

TOOLS.INI example

```
[nmake]
.SUFFIXES: .pli
COMPILE_OPTS = gonumber source
.pli.obj:
  PLI $*.pli ($(COMPILE_OPTS)
```

These lines in the `TOOLS.INI` file do the following:

Macros and inference rules in TOOLS.INI

- Add the `.pli` file extension to the list of extensions that can have inference rules.
- Define the COMPILE_OPTS macro as `gonumber source`.
- Define an inference rule to build `.obj` files from `.pli` source files.

Chapter 19. Improving performance

Improving performance

Many considerations for improving the speed of your program are independent of the compiler that you use and the platform on which it runs. This chapter, however, identifies those considerations that are unique to the workstation PL/I compilers and the code they generate.

Selecting compile-time options for optimal performance

The compile-time options you choose can greatly improve the performance of the code generated by the compiler. However, like most performance considerations, there are trade-offs associated with these choices. Fortunately, you can weigh the trade-offs associated with compile-time options without editing your source code because these options can be specified on the command line or in the environment variable IBM.OPTIONS.

If you want to avoid details, the least complex way to improve the performance of generated code is to specify the following (non-default) compile-time options:

```
PREFIX(NOFOFL)
IMPRECISE
OPT(2)
DFT(REORDER)
```

The first two options can affect the semantics of your program, but generally only do so in unusual situations. If you specify the first two options, your code is improved even when compiled with optimization turned off. By using these options, the compiler is also less likely to make errors.

The following sections describe, in more detail, performance improvements and trade-offs associated with specific compile-time options.

OPTIMIZE

You can specify the OPTIMIZE option to improve the speed of your program, otherwise, the compiler makes only basic optimization efforts.

Choosing OPTIMIZE(2) directs the compiler to generate code for better performance. Usually, the resultant code is shorter than when the program is compiled under NOOPTIMIZE. Sometimes, however, a longer sequence of instructions runs faster than a shorter sequence. This can occur, for instance, when a branch table is created for a SELECT statement where the values in the WHEN clauses contain gaps. The increased number of instructions generated in this case is usually offset by the execution of fewer instructions in other places.

IMPRECISE

When you select this option, the compiler generates smaller and faster sequences of instructions for floating-point operations. This can have a significant effect on the performance of programs that contain floating-point expressions, either separately or in loops.

However, when programs are compiled with the IMPRECISE option, floating-point exceptions might not be reported at the precise location where they occur. (This is especially true when the OPTIMIZE option is in effect.) In addition, floating-point operations can produce results that are not precisely IEEE conforming.

GONUMBER

Using this option results in a statement number table used for debugging. This added information can be extremely helpful when debugging, but including statement number tables increases the size of your executable file. Larger executable files can take longer to load.

By using one of the linker options, you can include the statement number tables in your executable during development to help with debugging. The /DEBUG (/DE) option directs the linker to include these tables in the executable file, so by not specifying /DE with the ILINK command, you can better control the size of your executable files. If the size of your executable file is a consideration, you can leave the tables out during production mode.

SNAP

When you use the SNAP option, the compiler generates extra instructions in the prolog and epilog code for every block. These instructions ensure that the run-time traceback messages (produced by PLIDUMP and the SNAP option on an ON statement) include all procedures that were active when the traceback was requested.

A trade-off of using the SNAP option and creating these additional instructions is that it can have a negative impact on the performance of your application. This is especially true for procedures that are called frequently.

RULES

When you use the RULES(IBM) option, the compiler supports scaled FIXED BINARY and, what is more important for performance, generates scaled FIXED BINARY results in some operations. Under RULES(ANS), scaled FIXED BINARY is not supported and scaled FIXED BINARY results are never generated. This means that the code generated under RULES(ANS) always runs at least as fast as the code generated under RULES(IBM), and sometimes runs faster.

For example, consider the following code fragment:

```
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.

Improving performance

When you use the RULES(LAXCTL) option, the compiler allows you to declare a CONTROLLED variable with a constant extent and then ALLOCATE it with a different extent, as in

```
DECLARE X BIT(1) CTL;

ALLOCATE X BIT(63);
```

However, this programming practice forces the compiler to assume that no CONTROLLED variable has constant extents, and consequently it will generate much less efficient code when these variables are referenced.

But, if you specify a constant extent for a CONTROLLED variable only when it will always have that length (or bound), then you will get much better performance if you specify the option RULES(NOLAXCTL).

PREFIX

This option determines if selected PL/I conditions are enabled by default. The default suboptions for PREFIX are set to conform to the PL/I language definition. However, overriding the defaults can have a significant effect on the performance of your program. The default suboptions are:

CONVERSION
INVALIDOP
FIXEDOVERFLOW
OVERFLOW
INVALIDOP
NOSIZE
NOSTRINGRANGE
NOSTRINGSIZE
NOSUBSCRIPTRANGE
UNDERFLOW
ZERODIVIDE

By specifying the SIZE, STRINGRANGE, STRINGSIZE, or SUBSCRIPTRANGE suboptions, the compiler generates extra code that helps you pinpoint various problem areas in your source that would otherwise be hard to find. This extra code, however, can slow program performance significantly.

CONVERSION

When you disable the CONVERSION condition, some character-to-numeric conversions are done inline and without checking the validity of the source. Therefore, specifying NOCONVERSION also affects program performance.

FIXEDOVERFLOW

On some platforms, the FIXEDOVERFLOW condition is raised by the hardware and the compiler does not need to generate any extra code to detect it. With personal computers, however, the hardware does not raise this condition so the compiler must

generate extra code. This extra code can negatively impact the performance of your program; and unless your program requires (or expects) this condition to be raised, specify PREFIX(NOFIXEDOVERFLOW) to improve performance.

DEFAULT

Using the DEFAULT option, you can select attribute defaults. As is true with the PREFIX option, the suboptions for DEFAULT are set to conform to the PL/I language definition. Changing the defaults in some instances can affect performance. The default suboptions are:

IBM
BYADDR
RETURNS(BYVALUE)
NONCONNECTED
DESCRIPTOR
ORDER
ASSIGNABLE LINKAGE(OPTLINK)
ASCII
IEEE
NATIVE
NODIRECTED
NOINLINE

The IBM/ANS, ASSIGNABLE/NONASSIGNABLE, and DIRECTED/NODIRECTED suboptions have no effect on program performance. All of the other suboptions can affect performance to varying degrees and, if applied inappropriately, can make your program invalid.

BYADDR or BYVALUE

When the DEFAULT(BYADDR) option is in effect, arguments are passed by reference (as required by PL/I) unless an attribute in an entry declaration indicates otherwise. As arguments are passed by reference, the address of the argument is passed from one routine (calling routine) to another (called routine) as the variable itself is passed. Any change made to the argument while in the called routine is reflected in the calling routine when it resumes execution.

Program logic often depends on passing variables by reference. However, passing a variable by reference can hinder performance in two ways:

1. Every reference to that parameter requires an extra instruction.
2. Since the address of the variable is passed to another routine, the compiler is forced to make assumptions about when that variable might change and generate very conservative code for any reference to that variable.

Consequently, you should pass parameters by value using the BYVALUE suboption whenever your program logic allows. Even if you use the BYADDR attribute to indicate that one parameter should be passed by reference, you can use the DEFAULT(BYVALUE) option to ensure that all other parameters are passed by value.

If a procedure receives and modifies only one parameter that is passed by BYADDR, consider converting the procedure to a function that receives that parameter by value. The function would then end with a RETURN statement containing the updated value of the parameter.

Procedure with BYADDR parameter:

```
a: proc( parm1, parm2, ..., parmN );

  dcl parm1 byaddr  ...;
  dcl parm2 byvalue ...;
     ⋮
  dcl parmN byvalue ...;

  /* program logic */

end;
```

Faster, equivalent function with BYVALUE parameter:

```
a: proc( parm1, parm2, ..., parmN )
   returns( ... /* attributes of parm1 */ );

  dcl parm1 byvalue  ...;
  dcl parm2 byvalue ...;
     ⋮
  dcl parmN byvalue ...;

  /* program logic */

  return( parm1 );

end;
```

(NON)CONNECTED

The DEFAULT(NONCONNECTED) option indicates that the compiler assumes that any aggregate parameters are NONCONNECTED. References to elements of NONCONNECTED aggregate parameters require the compiler to generate code to access the parameter's descriptor, even if the aggregate is declared with constant extents.

The compiler does not generate these instructions if the aggregate parameter has constant extents and is CONNECTED. Consequently, if your application never passes nonconnected parameters, your code is more optimal if you use the DEFAULT(CONNECTED) option.

RETURNS(BYVALUE) or RETURNS(BYADDR)

When the DEFAULT(RETURNS(BYVALUE)) option is in effect, the BYVALUE attribute is applied to all RETURNS description lists that do not specify BYADDR. This means that these functions return values in registers, when possible, in order to produce the most optimal code.

(NO)DESCRIPTOR

The DEFAULT(DESCRIPTOR) option indicates that, by default, a descriptor is passed for any string, area, or aggregate parameter. However, the descriptor is used only if the parameter has nonconstant extents or if the parameter is an array with the NONCONNECTED attribute.

In this case, the instructions and space required to pass the descriptor provide no benefit and incur substantial cost (the size of a structure descriptor is often greater than size of the structure itself). Consequently, by specifying DEFAULT(NODESCRIPTOR) and using OPTIONS(DESCRIPTOR) only as needed on PROCEDURE statements and ENTRY declarations, your code runs more optimally.

(RE)ORDER

The DEFAULT(ORDER) option indicates that the ORDER option is applied to every block, meaning that variables in that block referenced in ON-units (or blocks dynamically descendant from ON-units) have their latest values. This effectively prohibits almost all optimizations on such variables. Consequently, if your program logic allows, use DEFAULT(REORDER) to generate superior code.

LINKAGE

This suboption tells the compiler the default linkage to use when the LINKAGE suboption of the OPTIONS attribute or option for an entry has not been specified.

The compiler supports various linkages, each with its unique performance characteristics. When you invoke an ENTRY provided by an external entity (such as an operating system), you must use the linkage previously defined for that ENTRY.

As you create your own applications, however, you can choose the linkage convention. The OPTLINK linkage is strongly recommended because it provides significantly better performance than other linkage conventions.

ASCII or EBCDIC

The DEFAULT(ASCII) option indicates that, by default, character data is held in native Intel style. When you specify the EBCDIC suboption, the compiler must generate extra instructions for most operations involving the input or output of character variables.

IEEE or HEXADEC

The DEFAULT(IEEE) option indicates that, by default, float data is to be held in native Intel style. When you specify the HEXADEC suboption, the compiler must execute significantly more instructions for most operations involving floating-point variables.

(NON)NATIVE

The DEFAULT(NATIVE) option indicates that, by default, fixed binary data, offset data, ordinal data, and the length prefix of varying strings are held in native Intel style. When you specify NONNATIVE, extra instructions are generated for operations involving those data types previously listed.

(NO)INLINE

The suboption NOINLINE indicates that procedures and begin blocks should not be inlined.

Inlining occurs only when you specify optimization.

Inlining user code eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer, resulting in faster code performance. Inlining produces the best results when the overhead for the function is nontrivial, for example, when functions are called within nested loops. Inlining is also beneficial when the inlined function provides additional opportunities for optimization, such as when constant arguments are used.

For programs containing many procedures that are not nested:

- If the procedures are small and only called from a few places, you can increase performance by specifying INLINE.
- If the procedures are large and called from several places, inlining duplicates code throughout the program. This increase in the size of the program might offset any increase of speed. In this case, you might prefer to leave NOINLINE as the default and specify OPTIONS(INLINE) only on individually selected procedures.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends. Ensure that you have enough stack space for the local storage of the inlined functions.

Summary of compile-time options that improve performance

In summary, the following options (if appropriate for your application) can improve performance:

OPTIMIZE(2)
IMPRECISE
NOSNAP
PREFIX(NOFIXEDOVERFLOW)
RULES(ANS NOLAXCTL)
DEFAULT with the following suboptions
    (BYVALUE
    RETURNS(BYVALUE)
    CONNECTED
    NODESCRIPTOR
    REORDER
    ASCII
    IEEE
    NATIVE
    LINKAGE(OPTLINK)

Coding for better performance

As you write code, there is generally more than one correct way to accomplish a given task. Many important factors influence the coding style you choose, including readability and maintainability. The following sections discuss choices that you can make while coding that potentially affect the performance of your program.

DATA-directed input and output

Using GET DATA and PUT DATA statements for debugging can prove very helpful. When you use these statements, however, you generally pay the price of decreased performance. This cost to performance is usually very high when you use either GET DATA or PUT DATA without a variable list.

Many programmers use PUT DATA statements in their ON ERROR code as illustrated in the following example:

```
on error
  begin;
    on error system;
      ⋮
    put data;
      ⋮
  end;
```

In this case, the program would perform more optimally by including a list of selected variables with the PUT DATA statement.

The ON ERROR block in the previous example contained an ON ERROR system statement before the PUT DATA statement. This prevents the program from getting caught in an infinite loop if an error occurs in the PUT DATA statement (which could occur if any variables to be listed contained invalid FIXED DECIMAL values) or elsewhere in the ON ERROR block.

Input-only parameters

If a procedure has a BYADDR parameter which it uses as input only, it is best to declare that parameter as NONASSIGNABLE (rather than letting it get the default attribute of ASSIGNABLE). If that procedure is later called with a constant for that parameter, the compiler can put that constant in static storage and pass the address of that static area.

This practice is particularly useful for strings and other parameters that cannot be passed in registers (input-only parameters that can be passed in registers are best declared as BYVALUE).

In the following declaration, for instance, the first parameter to `dosScanEnv` is an input-only CHAR VARYINGZ string:

```
dcl dosScanEnv entry( char(*) varyingz nonasgn byaddr,
                      pointer byaddr )
               returns( native fixed bin(31) optional )
               options( nodescriptor linkage(system) );
```

If this function is invoked with the string 'IBM.OPTIONS', the compiler can pass the address of that string rather than assigning it to a compiler-generated temporary storage area and passing the address of that area.

String assignments

When one string is assigned to another, the compiler ensures that:

- The target has the correct value even if the source and target overlap
- The source string is truncated if it is longer than the target.

This assurance comes at the price of some extra instructions. The compiler attempts to generate these extra instructions only when necessary, but often you, as the programmer, know they are not necessary when the compiler cannot be sure. For instance, if the source and target are based character strings and you know they cannot overlap, you could use the PLIMOVE built-in function to eliminate the extra code the compiler would otherwise be forced to generate.

In the example which follows, faster code is generated for the second assignment statement:

```
dcl based_Str   char(64) based( null() );
dcl target_Addr pointer;
dcl source_Addr pointer;

target_Addr->based_Str = source_Addr->based_Str;

call plimove( target_Addr, source_Addr, stg(based_Str) );
```

If you have any doubts about whether the source and target might overlap or whether the target is big enough to hold the source, you should not use the PLIMOVE built-in.

Loop control variables

Program performance improves if your loop control variables are one of the types in the following list. You should rarely, if ever, use other types of variables.

FIXED BINARY with zero scale factor
FLOAT
ORDINAL
HANDLE
POINTER
OFFSET

Performance also improves if loop control variables are not members of arrays, structures, or unions. The compiler issues a warning message when they are. Loop control variables that are AUTOMATIC and not used for any other purpose give you the optimal code generation.

Performance is decreased if your program depends not only on the value of a loop control variable, but also on its address. For example, if the ADDR built-in function is applied to the variable or if the variable is passed BYADDR to another routine.

PACKAGEs versus nested PROCEDUREs

Calling nested procedures requires that an extra “hidden parameter” (the backchain pointer) is passed. As a result, the fewer nested procedures that your application contains, the faster it runs.

To improve the performance of your application, you can convert a mother-daughter pair of nested procedures into level-1 sister procedures inside of a package. This conversion is possible if your nested procedure does not rely on any of the automatic and internal static variables declared in its parent procedures.

If procedure b in “Example with nested procedures” does not use any of the variables declared in a, you can improve the performance of both procedures by reorganizing them into the package illustrated in “Example with packaged procedures.”

Example with nested procedures

```
a: proc;

  dcl (i,j,k) fixed bin;
  dcl ib      based fixed bin;
      ⋮
  call b( addr(i) );
      ⋮
  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;
end;
```

Example with packaged procedures

```
p: package exports( a );

  dcl ib      based fixed bin;

  a: proc;

    dcl (i,j,k) fixed bin;
      ⋮
    call b( addr(i) );
      ⋮
  end;

  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;

end p;
```

REDUCIBLE functions

REDUCIBLE indicates that a procedure or entry need not be invoked multiple times if the argument(s) stays unchanged, and that the invocation of the procedure has no side effects.

For example, a user-written function that computes a result based on unchanging data should be declared REDUCIBLE. A function that computes a result based on changing data, such as a random number or time of day, should be declared IRREDUCIBLE.

In the following example, *f* is invoked only once since REDUCIBLE is part of the declaration. If IRREDUCIBLE had been used in the declaration, *f* would be invoked twice.

```
dcl (f) entry options( reducible ) returns( fixed bin );

select;
  when( f(x) < 0 )
    ⋮
  when( f(x) > 0 )
    ⋮
  otherwise
    ⋮
end;
```

DEFINED versus UNION

The UNION attribute is more powerful than the DEFINED attribute and provides more function. In addition, the compiler generates better code for union references.

In the following example, the pair of variables b3 and b4 perform the same function as b1 and b2, but the compiler generates more optimal code for the pair in the union.

```
dcl b1 bit(31);
dcl b2 bit(16) def b1;

dcl
  1 * union,
    2 b3 bit(32),
    2 b4 bit(16);
```

Code that uses UNIONs instead of the DEFINED attribute is subject to less misinterpretation. Variable declarations in unions are in a single location making it easy to realize that when one member of the union changes, all of the others change also. This dynamic change is less obvious in declarations that use DEFINED variables since the declare statements can be several lines apart.

Named constants versus static variables

You can define named constants by declaring a variable with the VALUE attribute. If you use static variables with the INITIAL attribute and you do not alter the variable, you should declare the variable a named constant using the VALUE attribute. However, the compiler does not treat NONASSIGNABLE scalar STATIC variables as true named constants.

The compiler generates better code whenever expressions are evaluated during compilation, so you can use named constants to produce efficient code with no loss in readability. For example, identical object code is produced for the two usages of the VERIFY built-in function in the following example:

```
dcl numeric char  value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );
```

The following examples illustrate how you can use the VALUE attribute to get optimal code without sacrificing readability.

Example with optimal code but no meaningful names

```
dcl  x  bit(8) aligned;

select( x );
  when( '01'b4 )
     ⋮
  when( '02'b4 )
     ⋮
  when( '03'b4 )
     ⋮
end;
```

Coding for better performance

Example with meaningful names but not optimal code

```
dcl (  a1  init( '01'b4)
      ,a2  init( '02'b4)
      ,a3  init( '03'b4)
      ,a4  init( '04'b4)
      ,a5  init( '05'b4)
    ) bit(8) aligned static nonassignable;

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
     ⋮
  when( a2 )
     ⋮
  when( a3 )
     ⋮
end;
```

Example with optimal code AND meaningful names

```
dcl (  a1  value( '01'b4)
      ,a2  value( '02'b4)
      ,a3  value( '03'b4)
      ,a4  value( '04'b4)
      ,a5  value( '05'b4)
    ) bit(8);

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
     ⋮
  when( a2 )
     ⋮
  when( a3 )
     ⋮
end;
```

Avoiding calls to library routines

The bitwise operations (prefix NOT, infix AND, infix OR, and infix EXCLUSIVE OR) are often evaluated by calls to library routines. These operations are, however, handled without a library call if either of the following conditions is true:

- Both operands are bit(1)
- Both operands are aligned bit(8n) where n is a constant.

For certain assignments, expressions, and built-in function references, the compiler generates calls to library routines. If you avoid these calls, your code generally runs faster.

To help you determine when the compiler generates such calls, the compiler generates a message whenever a conversion is done using a library routine. The conversions done with code generated inline are shown in Table 29 on page 373.

Table 29. Conditions under which conversions are handled inline

Target	Source	Condition
fixed bin(p1,q1)	fixed bin(p2,q2)	always
	float(p2)	if SIZE is disabled
	bit(1)	always
	bit(n) aligned	if n is known and n ≤ 31
	char(1)	if CONV is disabled
	pic'(n)9'	if n ≤ 6
	pic'(n)Z(m)9'	if n + m ≤ 6
fixed dec(p1,q1)	fixed dec(p2,q2)	done using an especially fast library routine
float(p1)	fixed bin(p2,q2)	always
	float(p2)	always
	bit(1)	always
	bit(n) aligned	if n is known and n ≤ 31
	char(1)	if CONV is disabled
	pic'(n)9'	if n ≤ 6
	pic'(n)Z(m)9'	if n + m ≤ 6
pictured fixed	pictured fixed	if pictures match
pictured float	pictured float	if pictures match
char	char nonvarying	always
	char varying	always
	char varyingz	always
	pictured fixed	always
	pictured float	always
	pictured char	always
pictured char	pictured char	if pictures match
bit(1) nonvarying	bit(1) nonvarying	always
bit(n) nonvarying	bit(m) nonvarying	see note

Note: If all of the following apply:
1) source and target are byte-aligned
2) n and m are known
3) mod(m,8)=0 or n=m or source is a constant
4) mod(n,8)=0 or target is a scalar with STATIC, AUTOMATIC, or CONTROLLED attributes

Many string-handling built-in functions are evaluated through calls to library routines, but some are handled without a library call. Table 30 on page 374 lists these built-in functions and the conditions under which they are handled inline.

Coding for better performance

Table 30. Conditions under which string built-in functions are handled inline

String function	Comments and conditions
BOOL	When the third argument is a constant. The first two arguments must also be either both bit(1) or both aligned bit(n) where n is 8, 16 or 32. The function is also handled inline if it can be reduced to a bitwise infix operation and both arguments are aligned bit.
COPY	When the first argument has type character.
EDIT	When the first argument is REAL FIXED BIN, the SIZE condition is disabled, and the second argument is a constant string consisting of all 9's.
HIGH	Always
INDEX	When only two arguments are supplied and they have type character.
LENGTH	Always
LOW	Always
MAXLENGTH	Always
SEARCH	When only two arguments are supplied and they have type character.
SEARCHR	When only two arguments are supplied and they have type character.
SUBSTR	When STRINGRANGE is disabled.
TRANSLATE	When the second and third arguments are constant.
TRIM	When only one argument is supplied and it has type character.
UNSPEC	Always
VERIFY	When only two arguments are supplied and they have type character.
VERIFYR	When only two arguments are supplied and they have type character.

Chapter 20. Using user exits

Using the compiler user exit

PL/I provides a number of user exits that allow you to customize the PL/I product to suit your needs. The workstation PL/I products supply default exits and the associated source files.

If you want the exits to perform functions that are different from those supplied by the default exits, we recommend that you modify the supplied source files as appropriate.

The types of files provided include:

- PL/I source files with the extension PLI that are located in `\ibmpli\samples`.
- PL/I include files with the extension CPY that are located in `\ibmpli\include`. When compiling the user exits, make sure to set the INCLUDE or IBM.SYSLIB environment variables so that the CPY files can be found.
- Linker definition files with the extension DEF that are located in `\ibmpli\samples`.
- Control files (if applicable to the exit) with the extension INF that are located in `\ibmpli\samples`. When using the user exits, make sure the directory containing the INF files is specified using the appropriate environment variables (usually DPATH).

Using the compiler user exit

At times, it is useful to be able to tailor the compiler to meet the needs of your organization. For example, you might want to suppress certain messages or alter the severity of others. You might want to perform a specific function with each compilation, such as logging statistical information about the compilation into a file.

A compiler user exit handles this type of functions. With PL/I, you can write your own user exit or use the exit provided with the product, either `'as is'` or slightly modified depending on what you want to do with it. The purpose of this chapter is to describe:

- Procedures that the compiler user exit supports
- How to activate the compiler user exit
- IBMUEXIT, the IBM-supplied compiler user exit
- Requirements for writing your own compiler user exit.

Procedures performed by the compiler user exit

The compiler user exit performs three specific procedures:

- Initialization
- Interception and filtering of compiler messages
- Termination

As illustrated in Figure 32, 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.

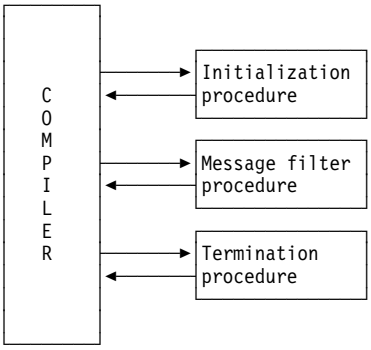


Figure 32. PL/I compiler user exit procedures

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 379.
- 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 381, “Writing the message filtering procedure” on page 381, and “Writing the termination procedure” on page 382.

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 50.

The EXIT compile-time option allows you to specify a user-option-string which specifies the message control file. If you do not specify a string, IBMUEXIT.INF is used (see “Modifying IBMUEXIT.INF” on page 378) but you have to tell the computer where to find it. The default behavior, provided you do not change the IBMUEXIT.PLI sample program, is that the compiler looks for IBMUEXIT.INF in the current directory first and then in the directories specified in DPATH.

The user-option-string is passed to the user exit functions in the global control block which is discussed in “Structure of global control blocks” on page 379. Please refer to the field “Uex_UIB_User_char_str” in the section “Structure of global control blocks” on page 379 for additional information.

The IBM-supplied compiler exit, IBMUEXIT

IBM supplies you with the sample compiler user exit, IBMUEXIT, which filters messages for you. It monitors messages and, based on the message number that you specify, suppresses the message or changes the severity of the message.

There are several files that comprise IBMUEXIT:

Using the compiler user exit

IBMUEXIT.PLI
Contains the PL/I source code.

IBMUEXIT.DLL
Executable DLL of IBMUEXIT.PLI. In order to build this file, issue the following commands from the command line:

OS/2 On OS/2:

```
pli ibmuexit
ilink /dll ibmuexit.obj ibmuexit.def
```

WIN On Windows:

```
pli ibmuexit
ilib /geni ibmuexit.def
ilink /dll ibmuexit.obj ibmuexit.exp
```

IBMUEXIT.DEF
DEF file that is used to build IBMUEXIT.DLL.

IBMUEXIT.INF
Control file that specifies filtering of messages.

The PLI source file is provided for your information and modification. The INF control file contains the message numbers that should be monitored, and tells IBMUEXIT what actions to take for them. The executable module reads the INF control file, and either ignores the message or changes its severity.

Customizing the compiler user exit

As was mentioned earlier, you can write your own compiler user exit or simply modify IBMUEXIT.PLI. In either case, the name of the executable file for the compiler user exit must be IBMUEXIT.DLL.

This section describes how to:

- Modify IBMUEXIT.INF for customized message filtering
- Create your own compiler user exit

Modifying IBMUEXIT.INF

Rather than spending the time to write a completely new compiler user exit, you can modify the sample program, IBMUEXIT.INF.

Edit the INF file to indicate which message numbers you want to suppress, and which message number severity levels you would like changed. A sample IBMUEXIT.INF file is shown in Figure 33.

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

Figure 33. Example of an IBMUEXIT.INF file

The first two lines are header lines and are ignored by IBMUEXIT. The remaining lines contain input separated by a variable number of blanks.

Each column of the file is relevant to the compiler user exit:

- The first column must contain the letters 'IBM' in single quotes, which is the message prefix.
- The second column contains the four digit message number.
- The third column shows the new message severity. Severity -1 indicates that the severity should be left as the default value.
- The fourth column indicates whether or not the message is to be suppressed. A '1' indicates the message is to be suppressed, and a '0' indicates that it should be printed.
- The comment field, found in the last column, is for your information, and is ignored by IBMUEXIT.

Writing your own compiler exit

To write your own user exit, you can use IBMUEXIT (provided as one of the sample programs with the product) as a model. As you write the exit, make sure it covers the areas of initialization, message filtering, and termination.

Structure of global control blocks

The global control block is passed to each of the three user exit procedures (initialization, filtering, and termination) whenever they are invoked. The following code and accompanying explanations describe the contents of each field in the global control block.

Using the compiler user exit

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

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

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

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

    2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs   */
    2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs   */

    2 Uex_UIB_Exit_Routs,                    /* exit entries set at
                                                initialization      */
      3 ( Uex_UIB_Termination,
          Uex_UIB_Message_Filter,            /* call for each msg   */
          *, *, *, * )
        limited entry (
            *,                               /* to Uex_UIB          */
            *,                               /* to a request area   */
         );
```

Data Entry Fields

- **Uex_UIB_ Length**: Contains the length of the control block in bytes. The value is `storage (Uex_UIB)`.
- **Uex_UIB_Exit_token**: Used by the user exit procedure. For example, the initialization may set it to a data structure which is used by both the message filter, and the termination procedures.
- **Uex_UIB_User_char_str**: Points to an optional character string, if you specify it. For example, in `pli filename (EXIT ('string'))...fn` can be a character string up to thirty-one characters in length.
- **Uex_UIB_char_len**: Contains the length of the string pointed to by the `User_char_str`. The compiler sets this value.
- **Uex_UIB_Filename_str**: Contains the name of the source file that you are compiling, and includes the drive and subdirectories as well as the filename. The compiler sets this value.
- **Uex_UIB_Filename_len**: Contains the length of the name of the source file pointed to by the `Filename_str`. The compiler sets this value.
- **Uex_UIB_return_code**: Contains the return code from the user exit procedure. The user sets this value.
- **Uex__UIB_reason_code**: Contains the procedure reason code. The user sets this value.

- **Uex_UIB_Exit_Routs**: Contains the exit entries set up by the initialization procedure.
- **Uex_UIB_Termination**: Contains the entry that is to be called by the compiler at termination time. The user sets this value.
- **Uex_UIB_Message_Filter**: Contains the entry that is to be called by the compiler whenever a message needs to be generated. The user sets this value.

Writing the initialization procedure

Your initialization procedure should perform any initialization required by the exit, such as opening files and allocating storage. The initialization procedure-specific control block is coded as follows:

```
Dcl 1 Uex_ISA native based( null() ),
    2 Uex_ISA_Length_fixed bin(31);   /* storage(Uex_IS    A) */
```

The global control block syntax for the initialization procedure is discussed in the section “Structure of global control blocks” on page 379.

Upon completion of the initialization procedure, you should set the return/reason codes to the following:

0/0 Continue compilation

4/n Reserved for future use

8/n Reserved for future use

12/n Reserved for future use

16/n Abort compilation

Writing the message filtering procedure

The message filtering procedure permits you to either suppress messages or alter the severity of messages. You can increase the severity of any of the messages but you can only decrease the severity of “WARNING” (severity code 4) messages to “INFORMATIONAL” (severity code 0) messages.

The procedure-specific control block contains information about the messages. It is used to pass information back to the compiler indicating how a particular message should be handled.

The following is an example of a procedure-specific message filter control block:

```
Dcl 1 Uex_MFA native based( null() ),
      2 Uex_MFA_Length   fixed bin(31),

      2 Uex_MFA_Facility_Id  char(3),        /* of component writing
                                                message             */
      2 *                    char(1),
      2 Uex_MFA_Message_no   fixed bin(31),
      2 Uex_MFA_Severity     fixed bin(15),
      2 Uex_MFA_New_Severity fixed bin(15);  /* set by exit proc    */
```

Data Entry Fields

- **Uex_MFA_Length**: Contains the length of the control block in bytes. The value is storage (Uex_MFA).
- **Uex_MFA_Facility_Id**: Contains the ID of the facility; in this case, the ID is IBM. The compiler sets this value.
- **Uex_MFA_Message_no**: Contains the message number that the compiler is going to generate. The compiler sets this value.
- **Uex_MFA_Severity**: Contains the severity level of the message; it can be from one to fifteen characters in length. The compiler sets this value.
- **Uex_MFA_New_Severity**: Contains the new severity level of the message; it can be from one to fifteen characters in length. The user sets this value.

Upon completion of the message filtering procedure, set the return/reason codes to one of the following:

0/0 Continue compilation, output message

0/1 Continue compilation, do not output message

4/n Reserved for future use

8/n Reserved for future use

16/n Abort compilation

Writing the termination procedure

You should use the termination procedure to perform any cleanup required, such as closing files. You might also want to write out final statistical reports based on information collected during the error message filter procedures and the initialization procedures.

The termination procedure-specific control block is coded as follows:

```
Dcl 1 Uex_ISA native based,
    2 Uex_ISA_Length_fixed bin(31); /* storage(Uex_ISA)     */
```

The global control block syntax for the termination procedure is discussed in “Structure of global control blocks” on page 379. Upon completion of the termination procedure, set the return/reason codes to one of the following:

0/0	Continue compilation
4/n	Reserved for future use
8/n	Reserved for future use
12/n	Reserved for future use
16/n	Abort compilation

Using the CICS run-time user exit

One of the key functions of the CICS run-time exit, CEEFXITA, is to let you control whether or not the CICS Dynamic Transaction Backout (DTB) occurs when PL/I transactions fail. The CICS run-time exit is driven immediately before and immediately after the invocation of each PL/I program within a transaction under CICS. Each time the exit is called, the typed structure CXIT (part of the include file IBMVCXT.INC) is used for communication between the PL/I run-time and the exit.

It is strongly recommended that you review and modify (if necessary) the user exit.

This structure contains information pertinent to the PL/I program, including:

- The reason for invocation (initialization or termination) of the exit.
- A reason code which indicates how the program terminated when invoked after program termination.
- Pointers to key CICS control blocks.

Prior to program invocation

When the exit is invoked before the invocation of the PL/I program, the exit can tell the PL/I run-time to bypass the program invocation. In this case, DTB occurs if necessary.

During this invocation of the exit, other functions (such as interrogation of or setting of run-time options) cannot be performed.

The IBM supplied exit merely returns allowing the PL/I program invocation to proceed.

After program termination

When the exit is invoked after the PL/I program invocation, the exit can examine the reason for program termination and can request DTB. The termination reason code indicates why the program ended. The file IBMVCXT.INC contains detailed information.

In this case, the IBM supplied exit requests DTB if:

- The PL/I program return code (set via PLIRETC) is non-zero
- The reason for termination is anything other than normal termination

Using data conversion tables

Modifying CEEFXITA

The following source files are supplied:

CEEFXITA.PLI
PL/I source code.

To recompile the exit, set the INCDIR compile-time option to include the directory for IBMVCXT.INC. Enter the following command at the command line:

```
pli CEEFXITA
```

IBMVCXT.INC
CXIT typed structure and other interface information.

CEEFXITA.DLL
Executable DLL.

To rebuild this DLL, issue the following command from the command line:

```
ilink /dll ceefxita.obj ceefxita.def
```

CEEFXITA.DEF
DEF file used to build CEEFXITA.DLL.

Using data conversion tables

The routines that the compiler, preprocessor, library, and debugger use to convert from ASCII to EBCDIC and from EBCDIC to ASCII are found in DLL files.

OS/2 For OS/2, the routines are found in these two files:

- ibmostb.dll (non-multithreading)
- ibmomtb.dll (multithreading)

WIN For Windows, the routines are found in these two files:

- ibmwstb.dll (non-multithreading)
- ibmwmtb.dll (multithreading)

The source for these routines, including the tables that they use, is shipped with the product so that you can use different tables if necessary. You might want to replace the tables if files are translated from EBCDIC to ASCII as you download them using a table different than the one we ship.

The names of the conversion routines are IBMPBE2A (EBCDIC to ASCII) and IBMPBA2E (ASCII to EBCDIC). Do not change the names of the files shipped with the product.

Definition files are also supplied with the product:

OS/2 For OS/2, the definition files are:

- ibmostb.def
- ibmomtb.def

WIN For Windows, the definition files are:

- ibmwstb.def
- ibmwmtb.def

You should use these definition files when creating the corresponding DLLs.

Chapter 21. Building dynamic link libraries

Dynamic linking is the process of resolving external references using dynamic link libraries (DLLs). Some advantages of dynamic linking are:

- Reduced memory requirements
- Simplified application modification
- Flexible software support
- Transparent migration of function
- Multiple programming language support
- Application-controlled memory usage.

DLLs are typically used to provide common functions that can be used by a number of applications. An application using a DLL can use either load-time dynamic linking or run-time dynamic linking.

You can dynamically link with the supplied run-time DLLs, as well as with your own DLLs. The following steps for creating and using a dynamic link library are described in this chapter:

- Creating the source files for a DLL
- Creating a module definition file (.DEF) for the DLL
- Compiling the source files and linking the resulting object files to build a DLL file
- Writing a module definition file to use when linking the external module that identifies what is in the DLL.

Each section contains a relevant example from the sample program SORT.PLI, which is packaged with the compiler.

Creating DLL source files

To build a DLL, you must first create source files containing the data or routines that you want to include in your DLL. No special file extension is required for DLL source files.

Each routine that you want to export from the DLL (that is, a routine that you plan to call from other executable modules or DLLs) must be an external routine, either by default or by being qualified with the external keyword.

Compiling your DLL source

You can compile your source files to create a DLL in the same way that you would compile any other file (using the PLI command) with one exception—you must compile at least one file with the DLLINIT option. You can compile every routine in a DLL with the DLLINIT option; however, no routine compiled with DLLINIT can be linked into an EXE.

You might also want to compile your programs with the option XINFO(DEF). This option creates a .DEF file for each program. These .DEF files are essential to preparing to link your DLL.

Preparing to link your DLL

When you link your DLL, you must tell the linker what names are to be exported out of the DLL. The steps you take to do this are different under OS/2 and Windows.

Specifying exported names under OS/2

OS/2 Under OS/2, you tell the linker what parts are exported using a .DEF file.

- If the names exported from your DLL are all defined in one .OBJ, you can use the .DEF file created by the compiler when you specify the XINFO(DEF) option. If that .OBJ contains external names that you do not want exported out of the DLL, you must delete them from the exports list built by the compiler.
- If the names come from more than one .OBJ, you can build the .DEF you want by merging the exports list from the .DEF files for all of the object files' exporting names. Again, if these object files contain external names that you do not want exported out of the DLL, you must delete them from the exports lists built by the compiler.

Your .DEF also contains several other statements. For a complete description of these statements and for more information about module definition files, refer to the OS/2 Developer's Toolkit documentation.

Specifying exported names under Windows

WIN Under Windows, you tell the linker what parts are exported using an .EXP file. The .EXP file is a binary file that is built by invoking `ilib` with the /GENI option and using either of the following as input:

- The .DEF file for the DLL
- All the .OBJ containing names to be exported by the DLL

Using .DEF files is preferable since it gives you control of exactly what is exported by the DLL. If you specify .OBJ names, all of the external names in the object files named are exported.

The following example shows a command you could use to create an .EXP file:

```
ilib /geni myliba.def
```

If you build the .EXP file by using a .DEF file, you can build the .DEF file as you would build it under OS/2 as described previously. The .DEF file created for Windows is different than the OS/2 .DEF file:

- The Windows version contains only an EXPORTS statement
- The Windows version contains names that have been 'decorated'

The name 'decoration' depends on a routine's linkage, but if you use the .DEF files created by the compiler, you do not need to be concerned about this.

Linking your DLL

To link your DLL, use the following options and input files:

Linker options

- /dll,
- /out: followed by the name of your dll

Input files

- All of the OBJs comprising your DLL
- The .DEF or .EXP file specifying what is to be exported

OS/2 For example, to link `mydlla.obj` and `mydllb.obj` into `mydlla.dll`, issue the following link command on OS/2:

```
ilink /dll /out:mydlla.dll mydlla.obj mydllb.obj mydlla.def
```

WIN Under Windows, issue this link command:

```
ilink /dll /out:mydlla.dll mydlla.obj mydllb.obj mydlla.exp
```

Using your DLL

Once you have built your DLL, other routines in your application can access the variables and routines exported by that DLL using one of the following methods:

- A FETCH statement
- Linking with an import library

If your application accesses an element of a DLL using a FETCH statement, you do not need to take any special action when you link. Unless your application executes that FETCH statement, the DLL does not even need to exist.

If your application accesses an element of a DLL as if it were statically linked with that DLL, then the linker must be able to resolve the name of that element.

Under OS/2, the linker can resolve names from a DLL if you link with a .DEF that has an IMPORTS statement listing the names being imported and the DLLs that own those names. For example, if your application needs to link to the name A from DLL X and the name from B from DLL Y, it would link to a .DEF file that looked like this:

```
IMPORTS
  X.A
  Y.B
```

Under Windows (and OS/2), the linker can resolve names from a DLL if you link with a import library for that DLL. In fact, that is how the names of PL/I library routines are resolved. For example, when you link with `ibmws20i.lib`, you are linking with the import library for `ibmws20.dll`.

OS/2 Under OS/2, the import library is built by the IMPLIB application available in the OS/2 Developer's Toolkit documentation.

Sample program to build a DLL

WIN Under Windows, the import library for the DLL is built when you create the .EXP file when preparing to link the DLL.

Note: In order for the loader to find a DLL, the DLL must reside either in your current working directory or in one of the directories listed in the LIBPATH environment variable under OS/2 or in the PATH environment variable under Windows.

Sample program to build a DLL

The sample programs SORT.PLI and DRIVER1.PLI show how to build and use a DLL that contains three different sorting functions. These functions keep track of the number of swap and compare operations required to do the sorting.

The files for the sample program are:

SORT.PLI
The source file for the DLL.

SORT.DEF
The module definition file for the DLL.

DRIVER1.DEF
The module definition file for the executable.

EXTDCL.CPY
The user include file.

DRIVER1.PLI
The main program that uses SORT.DLL.

OS/2 If you installed the sample programs, these files are found in the \IBMPLI\SAMPLES\ directory.

Use the following sequence of commands to compile, link, and run the program:

1. `pli driver1`
2. `pli sort`
3. `ilink driver1.obj /stack:80000 driver1.def`
4. `ilink /dll /out:sort sort.obj sort.def`
5. `driver1`

WIN If you installed the sample programs, these files are found in the \IBMPLIW\SAMPLES\ directory.

Use the following sequence of commands to compile, link, and run the program:

1. `pli sort`
2. `ilib /geni sort.def`
3. `ilink /dll /out:sort.dll sort.obj sort.exp`
4. `pli driver1`
5. `ilink driver1.obj /stack:80000 sort.lib`
6. `driver1`

Using FETCH and RELEASE in your main program

The SAMPLES directory also contains DRIVER2.PLI which is a modified version of DRIVER1.PLI that uses FETCH and RELEASE statements to dynamically link the SORT.DLL routines at run time instead of at load time.

The main advantage of using this version of the DRIVER program is that you can control when the sort routines are brought into and released from memory. Using FETCH and RELEASE statements, however, might increase your program's execution time.

OS/2 Use the following sequence of commands to compile, link, and run this version of the DRIVER program under OS/2:

1. `pli driver2`
2. `pli sort`
3. `ilink driver2.obj /stack:80000`
4. `ilink /dll /out:sort sort.obj sort.def`
5. `driver2`

WIN Use the following sequence of commands to compile, link, and run this version of the DRIVER program under Windows:

1. `pli sort`
2. `ilib /geni sort.def`
3. `ilink /dll /out:sort.dll sort.obj sort.exp`
4. `pli driver2`
5. `ilink driver2.obj /stack:80000`
6. `driver2`

Exporting data from a DLL

The preceding discussion described how to export external entrys from a DLL. You can also export external data from a DLL. To export external data from a DLL, the data must be declared as RESERVED throughout your application. The following conditions must also apply:

- The DLL that exports a variable must name that variable in the RESERVES option of some package in that DLL.
- All DLLs and EXEs importing a variable from another DLL must also declare that variable as RESERVED(IMPORTED).

For example, to create a DLL exporting just the variable `datatab`, the following routine would be used:

```
*process dllinit langlvl(saa2);

  edata: package reserves( datatab );

    dcl datatab char(256) reserved external init( .... );
  end;
```

Exporting data from a DLL

To import `datatab` into a procedure outside this DLL, it would be declared as:

```
dcl datatab char(256) reserved(imported) external;
```

Chapter 22. Using IBM Library Manager on OS/2

Using ILIB on OS/2

Use the IBM Library Manager (also referred to as ILIB) to create and maintain libraries of object code.

Library files are given the extension of .LIB (as in MYLIB.LIB). High Performance File System (HPFS) files with names that end with `.LIB` (as in MYLIBRARYFILE.NEW.LIB) are also supported.

ILIB works with standard libraries and OS/2 import libraries. It does not work with dynamic link libraries (DLLs).

Use the ILIB utility to:

- Create a new library (standard only)
- Add, delete, or replace modules in a library (import or standard)
- Copy object modules in a library to object files (from import or standard)
- List the contents of a library (import or standard)

See “Running ILIB” for general information on running ILIB.

Running ILIB

Run ILIB by typing `ILIB` at the operating system prompt.

You can specify parameters in one of three ways:

1. Enter them directly on the command line
2. Respond to prompts
3. Put them in a text file called a response file and specify the file name after the ILIB command.

To enter more commands than can be conveniently entered on one line, type an ampersand (&) at the end of the line and press Enter to extend the command field to a new line. You can use the ampersand with all three input methods.

You can press Ctrl+C or Ctrl+Break at any time while running ILIB to return to the operating system. Interrupting ILIB before completion restores the library from a backup.

Notes:

1. When started, ILIB makes a backup copy of the original library in case it is interrupted or a mistake is made. Make sure you have enough disk space for both your original library and the modified copy.
2. The library must end with the extension .LIB. If an extension is not specified, the default extension, `.LIB`, will be appended. HPFS file names are supported. Hence, MYLIBRARYNAME.NEW.LIB is still a valid library. This implies that MYLIBRARYNAME.NEW refers to MYLIBRARYNAME.NEW.LIB.
3. If you enter an input library name and follow it immediately with a semicolon (;), ILIB performs a consistency check on the library and takes no other action.

Using the command line

You can specify all the input ILIB needs on the command line. The syntax of the command line is:

`ILIB [`*`options`*`] `*`inlibrary`*` [`*`commands`*`] [[,`*`listfile`*`] [, `*`outlibrary`*`]] [;]`

options	Options that affect the behavior of ILIB.
inlibrary	The input library to be created or modified.
commands	Commands used to add, delete, replace, copy, and move modules within the library.
listfile	The name for a listing file. If you don't specify a name, no file is created.
outlibrary	The output library created from the input library. If you don't specify an output library, your input library is replaced with the modified version.

Commas are used to separate commands and options. The semicolon (;) is used to mark the end of the command line.

Using ILIB prompts

If you don't provide input to ILIB on the command line, ILIB prompts you for the information it needs by displaying the following messages, one at a time:

PROMPT	ENTER
Library name	Name of the input library to be modified. If the library you specify does not exist, the following prompt appears: `Library does not exist. Create library? (y or n)`
Operation(s)	Commands to modify the library. If no operations are specified, the input library is unchanged.
List file	Name for a listing file. If no listing file is specified, no listing file is created.
New Library Name	Name of the output library to be created from the input library. If no output library is specified, ILIB modifies the input library.

Enter the same information that you would enter when using the ILIB command line. You can enter ILIB options at any prompt.

Notes:

- ILIB waits for you to respond to each prompt before displaying the next prompt. If you notice that you have entered an incorrect response to a previous prompt, press Ctrl+C or Ctrl+Break to exit ILIB and begin again.
- A file name must be entered at the **Library name:** prompt. To choose a default response for any of the other prompts, press Enter. To choose default responses for all remaining prompts, type a semicolon (;) and press Enter.

Using ILIB on OS/2

Using an ILIB response file

To provide input to ILIB with a response file, type:

```
LIB @responsefile;
```

The *responsefile* is the name of a file containing the same information that can be specified on the command line.

In a sense, a response file extends the command line to include everything in the response file. To split input to ILIB between the command line and a response file, put part of your input on the command line and specify a response file (preceding the response file name with the at sign (@)). The response file name can be any valid file. To use special characters such as a space or the @ symbol, the filename must be enclosed in quotes.

ILIB responds to input you place in a response file just as it does to input you enter on a command line or after a prompt. Using a newline character in the response file is the equivalent of pressing the Enter key after an ILIB prompt.

A response file uses one text line for each prompt. To extend an ILIB command to multiple lines, end each line except the last with an ampersand (&). Responses must appear in the same order as the prompts. If a response for one of the prompts does not appear, the default is used.

Use a response file for:

- Complex and long commands you type frequently.
- Strings of commands that exceed the limit for command line length.

Examples specifying ILIB parameters

The following examples show different methods for specifying parameters to ILIB.

The operations shown in each example create a new library, `NEWLIB.LIB`, and its listing file, `NEWLIB.LST`, from the existing `MYLIB.LIB` library. `MYLIB.LIB` is unchanged, but `NEWLIB.LIB` has these changes:

- The contents are not case sensitive.
- The module TIM is deleted.
- The object file `SIMON.OBJ` is appended as an object module with the name SIMON.
- The module KEHM is deleted and is replaced by a new KEHM which is appended after SIMON.
- The module LAM is copied into an object file named `LAM.OBJ`.

Command line method

At the operating system prompt, enter the following two lines.

```
LIB /I MYLIB, SIMON-TIM-+KEHM &
*LAM, NEWLIB.LST, NEWLIB;
```

ILIB prompts method

To have ILIB prompt you for input, enter ILIB with no parameters.

```
Library name: /I MYLIB
 Library does not exist.  Create library? (y or n)  y
Operations: +SIMON-TIM-+KEHM &
Operations: *LAM
List file: NEWLIB.LST
New Library Name: NEWLIB
```

Response file method

First, create a response file with the following contents.

```
/I MYLIB
+SIMON-TIM-+KEHM &
*LAM
NEWLIB.LST
NEWLIB
```

Then, assuming the name of the response file is `response.fil`, invoke ILIB with:

```
ILIB @response.fil;
```

The lines in the response file match the entries you would have made with the prompting method. Even the ampersand character (&), the continuation character, is used in the same way.

Creating a new library

To create a new library file, specify the name of the library file you want to create on the command line (or at the **Library name:** prompt when using ILIB prompts).

Note: A library file is automatically created if the library file name you specify is immediately followed by a command, comma, or semicolon. In this case, the prompt does not appear.

If the name you specify for the new library file already exists, ILIB assumes that you want to modify the existing file.

When you give the name of a file that does not currently exist without specifying any operations, ILIB displays the following prompt:

```
Library does not exist.  Create library? (y or n)
```

Type `y` to create the file; type `n` to terminate the ILIB run. If you specify an extension other than `.LIB`, the ILIB utility tries to append the `.LIB` extension to the entire file name. If you do not enter a library name, ILIB prompts you for one.

Modifying a library

You can use ILIB to alter the contents of any object code library. For example, if you work with high level language libraries, you may want to replace a standard routine with your own version of the routine. You may also want to add a new routine to the standard library so that your routine is available along with the standard routines.

To modify an existing library file, specify the name of the library file you want to modify on the ILIB command line (or at the **Library name:** prompt when using ILIB prompts).

In the *commands* field, enter one or more commands to add, delete, or replace modules in the input library. Each command consists of a command character immediately followed by the name of the module or object file. Note that the Add command can be used to combine libraries as well as to add object files to a library.

ILIB creates a backup file of the library being modified if it already exists. This backup file has the same name as the original library with a `.BAK` filename extension.

Related information:

- "Add command (+)" on page 400
- "Delete command (–)" on page 401
- "Replace command (–+)" on page 401

Copying object modules to object files

To copy a module from a library file to an object file, specify the name of the library file on the ILIB command line (or at the *Library name:* prompt when using ILIB prompts).

To move or copy object modules, use the *commands* field on the ILIB command line:

Command	Action
Copy (*)	Copy the module to an object file and retain the module in the library.
Move (-*)	Copy the module to an object file and delete the module from the library.

Listing the contents of a library

Listings give you the exact names of modules and public symbols, allowing you to inspect the contents within a library.

To generate a listing file, enter the following on the command line (or at the appropriate ILIB prompt):

- The name of the library file in the *inlibrary* field.
- The name of the listing file in the *listfile* field.

When generating a listing file, the amount of detail can be varied. The level of detail is specified with the following option:

```
/Listlevel:n
```

Three levels of the option are available, with level **1** being the default. It is the fastest to generate and contains the least amount of information. All modules are listed in order of occurrence. For each module, the level 1 option:

- Shows the size of each module, and each module's file offset within the library.
- Lists all the public symbols defined in the module.
- Lists all external symbols referenced by the module.

Level **2** contains all the information of level 1. In addition, for each external symbol, level 2 shows which module in the library (if any) contains the required public symbols for resolving at link time. This can be overridden if a module is linked to another module that already contains the symbol.

Level **3** contains all the information of level 2. In addition, Level 3 displays:

- The technical characteristics of the library.
- A dump of the extended dictionary. This is useful to determine which modules will be implicitly linked in whenever a particular module is linked.

Sample cross reference listing

```
LIB /LISTLEVEL:2 NEWLIB, NEWLIB.LST;
```

The command above directs ILIB to place a listing of the contents of `NEWLIB.LIB` into the file `NEWLIB.LST`. No path specification is given for `NEWLIB.LST`. By default, the file created is put in the current directory.

ILIB commands

ILIB commands are used to manipulate modules in a library. When you run ILIB, you can specify multiple commands in any order.

Each command consists of a one- or two-character command symbol immediately followed by the name of the module or file that is the subject of the command. The following example adds the LEMKE.OBJ object file to a library as LEMKE.

```
+LEMKE.OBJ
```

Command	Action
[+]	Adds an object file or library to a library
-	Deletes a module from a library
-+	Replaces a module in a library
*****	Copies a module from a library to an object file
-*	Moves a module (copies the module and then deletes it)

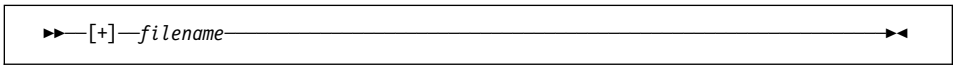
Using ILIB on OS/2

Notes:

- If you want to enter more commands than can be conveniently entered on one line, type an ampersand (&) and press Enter at the end of the line. This extends the command field to the next line.
- When processing commands, ILIB processes all copy commands first. ILIB processes the deletions next, and the additions last.
- ILIB never makes changes to your input library while it runs; it copies the library and makes changes to the copy. However, if you do not specify an output library, ILIB overwrites the input library with the modified copy at the end of normal processing. See “Using the command line” on page 395 for more information.

Add command (+)

Use the add command to add an object module or library to a library. The add command is issued by using the plus (+) sign or by leaving a blank space.



Adding an object module to a library

Type the name of the object file to be added immediately after the plus sign. The .OBJ extension may be omitted.

ILIB uses the base name of the object file as the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is CURSOR.

Object modules are always added to the end of a library file.

Combining two libraries

Specify the name of the library file to be added, including the .LIB extension, immediately after the plus sign (+). A copy of the contents of that library is added to the library file being modified. If both libraries contain a module with the same name, ILIB generates a warning message (LIB0003), and uses only the first module with that name.

ILIB adds the modules of the library to the end of the library being changed. The added library still exists as an independent library because ILIB copies the modules without deleting them.

Examples

This first example adds the file EFREM.OBJ to the library MYLIB.LIB.

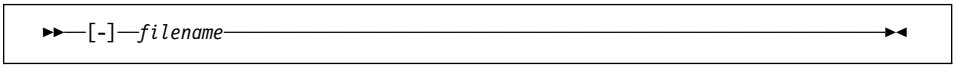
```
ILIB MYLIB +EFREM;
```

The following command adds the contents of the library KAREN.LIB to the library NEWLIB.LIB. The library KAREN.LIB is unchanged after this command is executed.

```
ILIB NEWLIB +KAREN.LIB;
```

Delete command (–)

Use the delete command (–) to delete an object module from a library. After the minus sign, specify the name of the module to be deleted. Module names do not have path names or extensions.



If the minus sign is omitted, the ADD command is assumed.

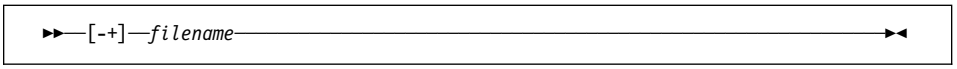
Example

The following command deletes the module EFREM from the library MYLIB.LIB.

```
ILIB MYLIB -EFREM;
```

Replace command (–+)

Use the replace command (–+) to replace a module in a library. Following the symbol, specify the name of the module to be replaced.



To replace a module, ILIB performs the following steps:

1. Deletes the existing module
2. Searches the current directory for the .OBJ file with the same file name as the deleted module
3. Appends to the library a copy of the object file with the original module name

Example

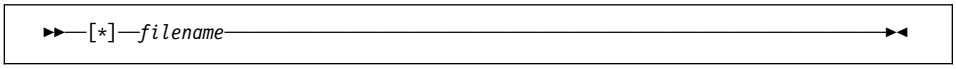
This example replaces the module EFREM in the MYLIB.LIB library with the contents of EFREM.OBJ from the current directory. The file EFREM.OBJ in the current directory is not altered.

```
LIB MYLIB -+EFREM;
```

Using ILIB on OS/2

Copy command (*)

Use the copy command (*) to copy a module from the library into an object file of the same name. The module remains in the library.



When ILIB copies the module to an object file, it adds the .OBJ extension to the module name and places the file in the current directory. If a file with this name already exists, ILIB overwrites the existing .OBJ file.

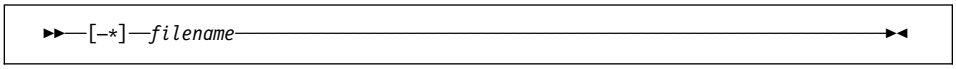
Example

This example copies the module EFREM from the MYLIB.LIB library to a file called EFREM.OBJ in the current directory. The module EFREM in MYLIB.LIB is not altered.

```
LIB MYLIB *EFREM;
```

Move command (–*)

Use the move command (–*) to copy an object module from the library file to an object file. The object module is then deleted from the library file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.



Example

This example moves the module KEELING from the MYLIB.LIB library to a file called KEELING.OBJ in the current directory. Upon completion of this process, MYLIB.LIB no longer contains the module KEELING.

```
LIB MYLIB -*KEELING;
```

ILIB options

Usage Notes:

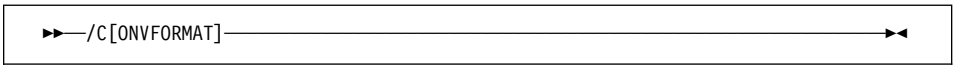
- Option characters are not case sensitive; /H and /h are equivalent.
- The characters in brackets can be omitted; /H and /HELP are equivalent.
- Unless otherwise specified, most options and commands need only the first letter of their names to be used.

The following is a summary of ILIB options:

Option	Action
/C[ONVFORMAT]	Convert input library to the new ILIB format
/H[ELP] or /? or ?	Display Help
/I[GNORECASE]	Turn case sensitivity off
/NOBA[CKUP]	Do not create backup copy of library
/NOE[XTDICTIONARY]	Do not generate extended dictionary
/NOI[GNORECASE]	Turn case sensitivity on
/NOL[OGO]	Suppress ILIB banner
/Q[UIET]	Suppress ILIB banner
/L[ISTLEVEL]	List current contents of library

/CONVFORMAT (convert to new format)

Use `/CONVFORMAT` to convert an existing library to the new ILIB format used by the linker.



ILIB only produces libraries in the new format; the PL/I linker accepts libraries in both the new format and in older formats, but link faster with libraries in the newer ILIB format.

To take advantage of the faster linking, you can convert a library to the new format using the `/CONVFORMAT` option.

Example

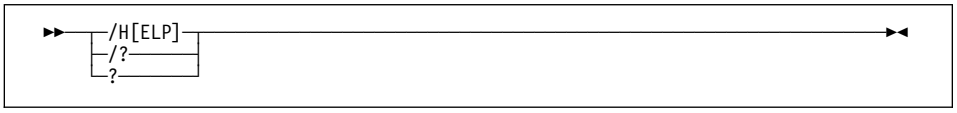
The following example converts the library `OLDLIB.LIB` from the previous LIB format to the new ILIB format:

```
ILIB OLDLIB.LIB /C
```

Using ILIB on OS/2

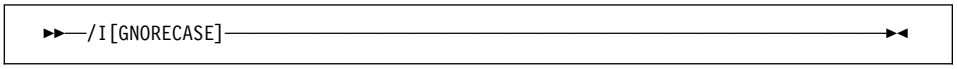
/HELP (display help)

Use /HELP to display a brief summary of ILIB syntax.



/IGNORECASE (turn case sensitivity off)

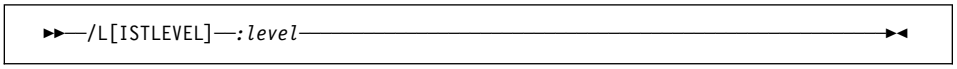
Use /IGNORECASE to turn off case sensitivity for symbols.



By default, case sensitivity is off. Use this option when you are combining a library that was created with case sensitivity on (using the /NOI option) with others that are not case sensitive. The resulting library is not case sensitive.

/LISTLEVEL (set detail level of listing)

Use /LISTLEVEL to set the detail level of an ILIB listing.



You can set the detail *level* as follows:

Level 1 Is the default. It is the fastest to generate and contains the least amount of information. All modules are listed in order of occurrence. For each module, the level 1 option:

1. Shows the relative position and size of each module.
2. Lists all the public symbols defined in the module, and their attributes.
3. Lists all external symbols which must be resolved at link time.

Level 2 Contains all the information of level 1. In addition, for each external symbol, level 2 shows which module in the library contains the required public symbols for resolving at link time. This can be overridden if a module is linked to another module that already contains the symbol.

Level 3 Contains all the information of level 2. In addition, Level 3 displays the technical characteristics of the library. This option also contains a dump of the extended dictionary. This is useful to determine which modules will be implicitly linked in whenever a particular module is linked in.

Note: Definitions with mangled names will be listed with the demangled form in brackets.

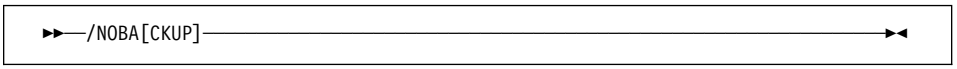
Sample Cross Reference Listing

This example directs ILIB to place a listing of the contents of NEWLIB.LIB into the file NEWLIB.LST. No path specification is given for NEWLIB.LST. By default, the file created is put in the current directory.

```
LIB /LISTLEVEL:2 NEWLIB, NEWLIB.LST;
```

/NOBACKUP (do not create backup)

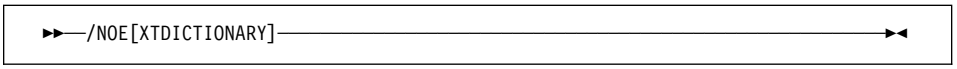
Use /NOBACKUP to prevent ILIB from creating a backup of the library.



By default, ILIB creates a backup of the library before it is modified.

/NOEXTDICTIONARY (do not generate extended dictionary)

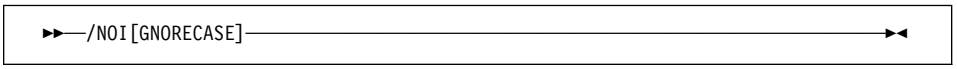
Use /NOEXTDICTIONARY to disable generation of the extended dictionary.



The extended dictionary is an optional part of the library that increases linking speed. However, using an extended dictionary requires more memory. The space reserved for the extended dictionary is limited to 64K, no more can be allocated. If ILIB reports an *out-of-memory* error, you may want to use this option. As an alternative, you can split large libraries into smaller libraries to use in linking.

/NOIGNORECASE (turn case sensitivity on)

Use /NOIGNORECASE to turn on case sensitivity.



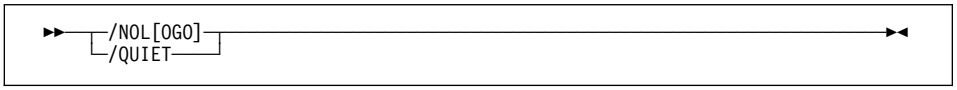
By default, case sensitivity is off (/I option). Using this option allows symbols that differ only in case, such as Sine and SINE, to be included as separate symbols in the same library.

Note that when you create a library with the /NOI option, ILIB *marks* the library internally to indicate that /NOI is in effect. If you combine multiple libraries, and any one of them is marked /NOI, then the output library is marked /NOI.

Using ILIB on OS/2

/NOLOGO|/QUIET (suppress banner)

Use /NOLOGO to suppress the ILIB copyright notice.



This option suppresses the banner message when ILIB is started. It can be used in batch files.

Chapter 23. Using IBM Library Manager on Windows

Using ILIB on Windows

Use the IBM Library Manager (also referred to as ILIB) to create and maintain libraries of object code, create import libraries and export object pairs, and generate module definition (`.def`) files. Using the ILIB utility, you can:

- Create a new library from a collection of objects
- Maintain a library
  - Add objects to an existing library
  - Delete objects from an existing library
  - Copy objects from an existing library
  - Replace objects in an existing library
- List the contents of a new or existing library
- Create import library/export object pairs from:
  - Module definition (`.def`) files
  - Objects generated from source files containing `#pragma export` and `_Export` statements
  - A combination of the above
- Generate module definition (`.def`) files from:
  - An existing DLL
  - Objects generated from source files containing `#pragma export` and `_Export` statements
  - A combination of the above

Running ILIB

Run ILIB by typing `ilib` at the command prompt.

You can specify parameters in the following ways:

1. Enter them directly on the command line
2. Use the ILIBenvironment variable
3. Put them in a text file, called a response file and specify the file name after the `ilib` command.
4. A combination of the above

You can press Ctrl+C or Ctrl+Break at any time while running ILIB to return to the operating system. Interrupting ILIB before completion restores the original library from a backup.

Notes:

1. When started, ILIB makes a backup copy of the original library in case it is interrupted or a mistake is made. Make sure you have enough disk space for both your original library and the modified copy.
2. The library must end with the extension `.lib`. If an extension is not specified, the default extension, `.lib`, will be appended. High Performance File System (HPFS) file names are supported. Hence, `mylibraryname.new.lib` is still a valid library.

Using the command line

You can specify all the input ILIB needs on the command line. The syntax of the command line is:

```
ilib [options] [libraries] [@responsefile] [objects]
```

Options	Options that affect the behavior of ILIB
Libraries	The input library to be created or modified
Response file	The name of a text file containing ILIB options
Objects	Commands used to add, delete, replace, copy, and move object modules within the library

The ILIB command line is a free format command line; that is, the input arguments can be specified any number of times, in any order. The only exception is the /FREEFORMAT option, which does have a position restriction. See "/FREEFORMAT" on page 417 for more information.

Note: For compatibility with the OS/2 release of ILIB, a fixed format command line is also supported. To use the fixed format command line, the /NOFREEFORMAT option must be specified immediately following `ilib` on the command line, or as the first parameter in the ILIB environment variable. The default command line format is free format.

For the purposes of this document, only the free format command line will be described in detail.

Using the ILIB environment variable

You can use the ILIB environment variable to specify any default ILIB options. When the `ilib` command is invoked, the environment variable will be parsed before the command line.

Use the `SET` command to give value to the ILIB environment variable. You can do this in the following ways:

Command line

When the `SET` command is used on the command line, the values you specify are in effect for only that session. They override values previously specified.

You can append the original value of the variable using %*variable*%. The following example would cause the ILIB environment variable to be set to the original value of the ILIB environment variable, with the /NOFREEFORMAT option specified ahead of any existing options.

```
SET ILIB=/FREEFORMAT %ILIB%
```

Windows NT control panel

Windows NT allows you to update environment variables and have them take effect immediately (that is, no reboot required) using the Windows NT Control Panel.

Using ILIB on Windows

To set the ILIB environment variable:

- Select the **Main** group by double-clicking on the **Main** icon.
- Select the **System** icon from the **Main** group by double-clicking on it.
- Enter `ILIB` in the **Variable** field.
- Enter the value for the ILIB environment variable in the **Value** field.
- Choose **Set**.

Windows 95 AUTOEXEC.BAT file

Windows 95 allows you to set environment variables in the `AUTOEXEC.BAT` file. Any environment variables set in this fashion are available in every user session.

Add a line to your `AUTOEXEC.BAT` file that sets the environment variable to the value you want. Consider the following example:

```
SET ILIB=/NOBACKUP
```

Because environment variables specified in your `AUTOEXEC.BAT` file are in effect for every session you start, this is a good place to specify options that you want to apply each time you invoke ILIB. However, after you make a change to your `AUTOEXEC.BAT` file, you must reboot your system to have the change take effect.

Using an ILIB response file

To provide input to ILIB with a response file, type:

`ilib @`*`responsefile`*

The *responsefile* is the name of a file containing the same information that can be specified on the command line.

Why use a response file?

Use a response file for:

- Complex and long commands you type frequently
- Strings of commands that exceed the limit for command line length.

A response file extends the command line to include everything in the response file. To split input to ILIB between the command line and a response file, put part of your input on the command line and specify a response file (preceding the response file name with the at sign (`@`)). No space can appear between the at sign and the file name.

The response file name can be any valid Windows file name. To use special characters in the file name, such as a space or the @ symbol, the file name must be enclosed in quotes.

ILIB responds to input you place in a response file just as it does to input you enter on a command line. Any newline characters that occur between arguments are treated as spaces. This allows you to extend an ILIB command to multiple lines.

Note: The options which specify which format command line to use (/FREEFORMAT or /NOFREEFORMAT) must be specified as the first parameter following `ilib` on the

command line or as the first parameter in the ILIB environment variable. They cannot be specified inside the response file.

Examples specifying ILIB parameters

The following examples show different methods for specifying parameters to ILIB.

The operations shown in each example create a new library, `newlib.lib`, and its listing file, `newlib.lst`, from the existing `mylib.lib` library. `mylib.lib` is unchanged, but `newlib.lib` has these changes:

- The module `text` is deleted
- The object file `root.obj` is appended as an object module with the name `root`
- The module `table` is deleted and is replaced by a new `table` which is appended after `root`
- The module `string` is copied into an object file named `string.obj`

Command Line Method

At the command line prompt, enter the following:

```
ilib /out:newlib.lib /list:newlib.lst mylib.lib /remove:text root table /extract:string
```

Response File Method

First, create a response file with the following contents:

```
/out:newlib.lib
/list:newlib.lst
mylib.lib
/remove:text
root table
/extract:string
```

Then, assuming the name of the response file is `response.fil`, invoke ILIB with:

```
ilib @response.fil
```

Controlling ILIB input

ILIB determines the format of any input files by examining the file contents. Most file formats can be identified by the file header information. If the format of an input file is not recognized and seems to contain only ASCII, it is assumed to be a module definition (`.def`) file.

ILIB allows you to place any extension you choose on a file and still have it dealt with correctly.

Controlling ILIB output

ILIB determines what output is to be produced by examining the options that you supply on the command line. The following options control ILIB output:

Using ILIB on Windows

Option	*Description*
/O[UT]:*filename*	A static library is produced.
/GEND[EF]:*filename*	A module definition (`.def`) file is produced. The short form, **/gd**, may also be used.
/GENI[MPLIB]:*filename*	An import library/export object pair is produced. The short form, **/gi**, may also be used.
/L[IST]:*filename*	A list file is produced.

If none of the above are specified, ILIB will determine what is to be produced, as follows:

- If a DEF file is input to ILIB, an import library/export object pair will be produced.

  Note: If there are no exported symbols, then no import library will be produced.

- If a library and/or object(s) are input to ILIB, a library combining them will be produced.

ILIB will allow you to generate a DEF file directly from a DLL. However, since the only information that a DLL has in it is the undecorated (exported) names, symbol decoration (calling convention) and type information (function or data) cannot be determined. ILIB will assume that all symbols exported from the DLL are _Optlink (the default linkage convention), unless an object file is provided that indicates otherwise.

The best way of using ILIB with a DLL is to use ILIB to create a DEF file using the **/gd** option. Edit the DEF file to change decorations, where appropriate, and then run the DEF file through ILIB using the **/gi** option to produce an import library/export object pair.

If an import library/export object pair is requested, and only a DLL is specified as input, ILIB will generate an error.

Controlling ILIB output

The following are examples showing how to control ILIB output.

Library

The following example creates create the library `newlib.lib` out of the objects in `text.obj` and `mylib.lib`.

```
ilib /out:newlib.lib text.obj mylib.lib
```

Note: Unless `newlib.lib` is specified as an input file, its contents will not be included in the library. If an output file already exists, and is not used as an input file, it will be replaced.

DEF File

This example creates the module definition file `winner.def` from the DLL `winner.dll`.

```
ilib /gd:winner.def winner.dll
```

Import Library/Export Oject Pair

The following example creates an import library named `winner.lib` and an export object named `winner.exp`. However, if no exported symbols are contained in `winner.def`, then `winner.lib` will not be produced.

```
ilib /gi winner.def
```

List File

The following example generates the list will generate the list file `mylib.lst`, based on the library `mylib.lib`, in the current directory.

```
ilib /list:mylib.lst mylib.lib
```

ILIB objects

ILIB objects are used to manipulate modules in a library. When you run ILIB, you can specify multiple objects in any order.

Each object consists of the ILIB command, followed by the name of the object module that is the subject of the command. Separate objects on the command line with a space or tab character.

Summary of ILIB objects

The following is a summary of ILIB objects on Windows.

Table 31. ILIB objects on Windows

Syntax	Description	Default	Page
filename	Add/replace the named object in the library	None	414
/E[XTRACT]:*obj*	Copy the named object into the current directory and overwrite it if it already exists	None	415
/R[EMOVE]:*obj*	Remove the named object from the list of objects to be placed in the output library	None	415

Notes:

1. ILIB objects are not case sensitive, so you can specify them in lower-, upper-, or mixed-case.

   You can also substitute a dash (-) for the slash (/) preceding the object. For example, -REMOVE:*filename* is equivalent to /REMOVE:*filename*.

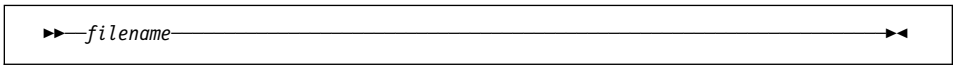
2. You can specify objects in either short or long form. For example, /R:*filename* and /RE:*filename* are equivalent to /REMOVE:*filename*.

3. The order of operations when processing the command line is left to right.

4. ILIB never makes changes to your input library while it runs. It copies the library and makes changes to the copy. If ILIB is interrupted, your original library will be restored.

   If you do not specify an output library, ILIB will not produce any output.

Add/Replace



The default action, when *filename* is specified on the command line without an associated object, is to add it to the library. If *filename* already exists in the library, it will be replaced.

Adding an Object Module to a Library

Type the name of the object file to be added on the command line. The `.obj` extension may be omitted.

ILIB uses the base name of the object file as the name of the object module in the library. For example, if the object file `cursor.obj` is added to a library file, the name of the corresponding object module is `cursor`.

Object modules are always added to the end of a library file.

Replacing an Object Module in a Library

Type the name of the object module to be replaced on the command line. The `.obj` extension may be omitted.

If the object module already exists in the library, ILIB will replace it with the new copy.

Combining Two Libraries

Specify the name of the library file to be added, including the `.lib` extension, on the command line. A copy of the contents of that library is added to the library file being modified. If both libraries contain a module with the same name, ILIB generates a warning message, and uses only the first module with that name.

ILIB adds the modules of the library to the end of the library being changed. The added library still exists as an independent library because ILIB copies the modules without deleting them.

Examples

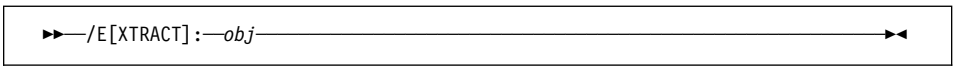
The following command adds the file `sample.obj` to the library `mylib.lib`. If `sample.obj` already exists in the library `mylib.lib`, ILIB will replace it.

```
ilib /out:mylib.lib mylib.lib sample.obj
```

This example adds the contents of the library `mylib.lib` to the library `newlib.lib`. The library `mylib.lib` is unchanged after this command is executed.

```
ilib /out:newlib.lib newlib.lib mylib.lib
```

/EXTRACT



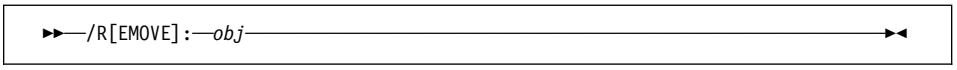
Use /EXTRACT to copy a module from the library into an object file of the same name. The module remains in the library.

When ILIB copies the module to an object file, it adds the `.obj` extension to the module name and places the file in the current directory. If a file with this name already exists, ILIB overwrites it.

Example The command above copies the module `sample` from the `mylib.lib` library to a file called `sample.obj` in the current directory. The module `sample` in `mylib.lib` is not altered.

```
ilib mylib.lib /extract:sample
```

/REMOVE



Use /REMOVE to delete an object module from a library. After /REMOVE, specify the name of the module to be deleted. Module names do not have path names or extensions.

Examples The following command deletes the module `sample` from the library `mylib.lib`.

```
ilib /out:mylib.lib mylib.lib /remove:sample
```

This next command copies `sample.obj` from the `mylib.lib` library to an object file in the current directory. Then `sample.obj` is deleted from the library.

```
ilib /out:mylib.lib mylib.lib /extract:sample /remove:sample
```

ILIB options

ILIB options affect the behavior of ILIB. When you run ILIB, you can specify multiple options in any order. The only exception is the /FREEFORMAT option, which has a position restriction.

Separate options on the command line with a space or tab character.

Using ILIB on Windows

Summary of ILIB options

The following is a summary of ILIB options on Windows.

Table 32. ILIB options on Windows

Syntax	Description	Default	Page
/?	Display help	None	416
/BA[CKUP] /NOBA[CKUP]	Back up the output file (if it exists) before overwriting it	/BA	417
/DEF:*def*	Specify the name of a `.def` file to use to get information about exported symbols and linker parameters	None	417
/F[REEFORMAT] /NOF[REEFORMAT]	Use the free format command line	/F	417
/GEND[EF]:*filename*	Generate a `.def` file	None	417
/GENI[MPLIB]:*filename*	Generate an import library	None	418
/H[ELP]	Display help	None	418
/L[IST]:*filename*	Generate a list file	None	418
/NOE[XTDICTIONARY] /EXTD[ICTIONARY]	Do not generate an extended dictionary in an OMF library	/EXTD	418
/O[UT]:*filename*	Specify the name of the output library	None	419
/Q[UIET], /NOL[OGO] /LO[GO], /NOQ[UIET]	Do not display the banner on startup	/LO	419
/W[ARN:*msgnum*,*msgnum*[,...]] /NOW[ARN:*msgnum*,*msgnum*[,...]]	Enable printing of warning message number *msgnum*	None	419

Notes:

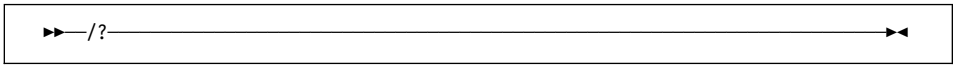
1. ILIB options are not case sensitive, so you can specify them in lower-, upper-, or mixed-case.

   You can also substitute a dash (-) for the slash (/) preceding the option. For example, -FREEFORMAT is equivalent to /FREEFORMAT.

2. You can specify options in either short or long form. For example, /F, /FR, and /FREE are equivalent to /FREEFORMAT.

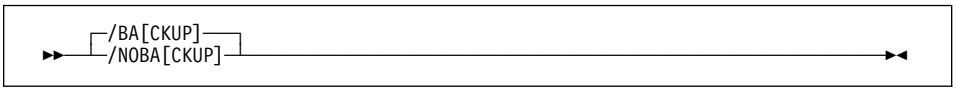
See below for detailed information on each ILIB option.

/?



Use /? to display a list of valid ILIB options. This option is equivalent to /HELP.

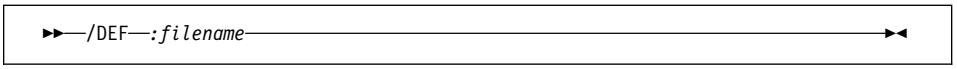
/BACKUP



Use /BACKUP to back up the output file (if it exists) before overwriting it.

ILIB uses the base name of the library as the name of the backup library, and then appends the .bak extension. For example, if the library being modified is `mylib.lib` and a backup is requested, ILIB will create `mylib.bak` in the current directory.

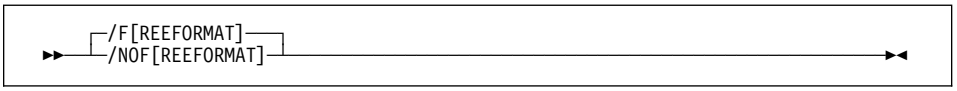
/DEF



Use /DEF to specify the name of the .def file to use to get information about exported symbols and linker parameters.

This option is not required, since ILIB will recognize .def files by their contents if they are placed with other input files on the command line.

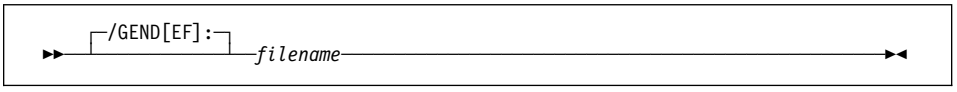
/FREEFORMAT



Use the /FREEFORMAT option to tell ILIB that you are using the free format command line. The free format command line allows you to specify ILIB input arguments any number of times, in any order.

Note: This option must be specified immediately following `ilib` on the command line, or as the first argument in the ILIB environment variable. If you don't specify either /FREEFORMAT or /NOFREEFORMAT, ILIB will default to the free format command line.

/GENDEF



Use the /GENDEF option to create a module definition (.def) file.

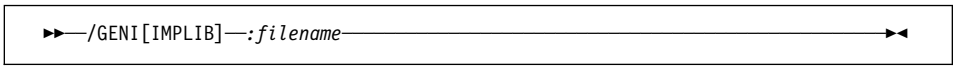
Example

The following command creates the module definition file `sample.def` from the DLL `sample.dll`.

```
ilib /gd:sample.def sample.dll
```

Using ILIB on Windows

/GI



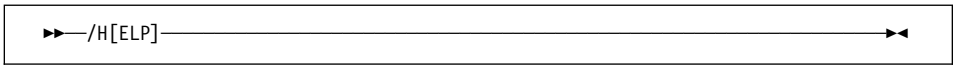
Use the /GENIMPLIB option to create an import library/export object pair.

Example

The command above will create an import library named `sample.lib` and an export object named `sample.exp` from the module definition file `sample.def`. However, if no exported symbols are contained, then `sample.lib` will not be produced.

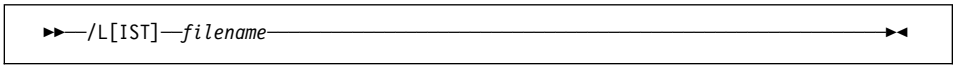
```
ilib /gi sample.def
```

/HELP



Use /HELP to display a list of valid ILIB options. This option is equivalent to /?.

/LIST



Use the /LIST option to generate a list file. If `filename` is not specified, ILIB will add the extension `.lst` to the input filename.

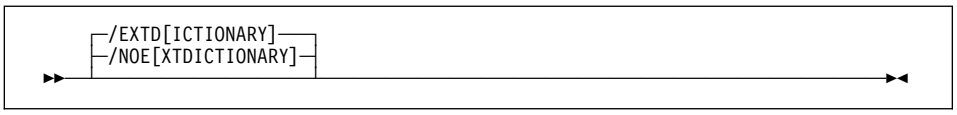
Example

The following command directs ILIB to place a listing of the contents of `mylib.lib` into the file `mylib.lst`. No path specification is given for `mylib.lst`. By default, the file created is put into the current directory.

```
ilib mylib /list:mylib.lst
```

Note: The /LISTLEVEL option is not supported in the Windows release of ILIB.

/NOEXT

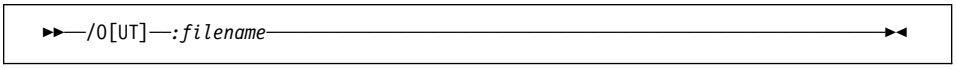


Use /NOEXTDICTIONARY to disable generation of the extended dictionary.

The extended dictionary is an optional part of the library that increases linking speed. However, using an extended dictionary requires more memory. The space reserved for the extended dictionary is limited to 64K. If ILIB reports an *out-of-memory* error, you

may want to use this option. As an alternative, you can split large libraries into smaller libraries to use in linking.

/OUT



Use the /OUT option to create or maintain an existing library. If *filename* is not specified, then the filename of the first input file specified on the command line will be used. ILIB will add the `.lib` extension.

Examples

The following command creates the library `newlib.lib` out of the objects in `mylib.lib` and `sample.obj`.

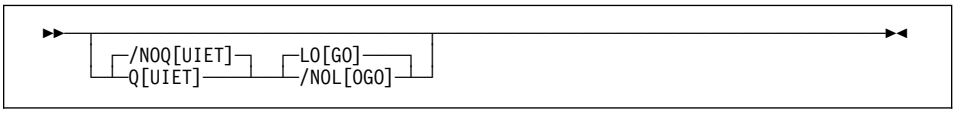
```
ilib /out:newlib.lib mylib.lib sample.obj
```

Note: Unless `newlib.lib` is specified as an input file, its contents are not included in the library.

This command creates the library `newlib.lib` by combining it with the library `mylib.lib`.

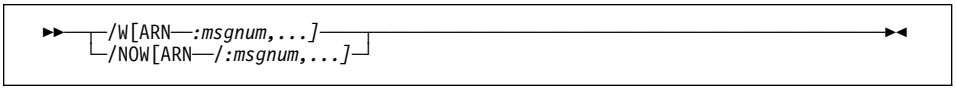
```
ilib /out newlib.lib mylib.lib
```

/QUIET



Use the /QUIET or /NOLOGO options to suppress the ILIB copyright notice.

/WARN



Use the /WARN option to enable printing of the message number specified in the *msgnum* parameter.

Chapter 24. Calling conventions

This chapter discusses the calling conventions used by VisualAge PL/I:

OPTLINK
SYSTEM
STDCALL
CDECL

The OPTLINK linkage convention (see “OPTLINK linkage” on page 423 for details) is also supported by VisualAge for C++ (OS/2 and Windows) and is the fastest method of calling PL/I procedures, C functions, or assembler routines. OPTLINK is not, however, standard for all OS/2 and Windows applications.

OS/2 The SYSTEM linkage convention, while slower, is standard for all OS/2 applications and is used for calling OS/2 APIs. For a description of the SYSTEM linking convention, see “SYSTEM linkage” on page 432.

WIN On Windows, specifying SYSTEM linkage is synonymous with STDCALL linkage and is implemented the same as STDCALL. The compiler, however, considers the names SYSTEM and STDCALL to be distinct and complains if you mix them. The STDCALL linking convention is described in “STDCALL linkage (Windows only)” on page 434.

You can specify the calling convention for all functions within a program using the LINKAGE suboption of the DEFAULT compile-time option. You can also use the LINKAGE option of the OPTIONS attribute to specify the linkage for individual functions.

Note: You cannot call a function using a different calling convention than the one with which it is compiled. For example, if a function is compiled with SYSTEM linkage, you cannot later call it specifying OPTLINK linkage.

Understanding linkage considerations

On OS/2, there are three primary linkages that the PL/I compiler supports: OPTLINK, CDECL, and SYSTEM. On Windows, there are also three: OPTLINK, CDECL, and STDCALL. As was mentioned ealier, on OS/2 all the system services use the SYSTEM linkage while on Windows, all the system services use the STDCALL linkage.

These linkages differ in their parameter passing conventions:

- The OPTLINK linkage is the only one that attempts to pass some parameters in registers; the other linkages pass all the parameters on the stack.
- The STDCALL linkage is the only one that makes the callee responsible for cleaning up the stack; the other linkages make the caller responsible.

So the SYSTEM linkage on OS/2 seems to match up with the CDECL linkage on Windows. However, the VisualAge PL/I compiler interprets any specification of the SYSTEM linkage as if the STDCALL linkage were intended. The VisualAge C compiler does the same.

Understanding linkage considerations

But, there is another important difference between OS/2 and Windows linkages. On Windows, all external names are decorated. If the external attribute does not specify a name, the name decoration depends on the linkage:

- Routines with the CDECL linkage have a '_' added as a prefix so that, for example, the name FUNKY would become _FUNKY.
- Routines with the OPTLINK linkage have a '?' added as a prefix so that, for example, the name FUNKY would become ?FUNKY.
- Routines with the STDCALL linkage have a '_' added as a prefix and a '@' followed by the bytes used by its parameters added as a suffix. For example, then, if the name FUNKY had two byvalue pointers or any two byaddr parameters, it would become _FUNKY@8.

One consequence of these name decorations is that if a caller of a routine specifies the wrong linkage for that routine, the program fails to link.

So far, the discussion of name decoration has applied only to routines for which the external attribute did not specify a name. It also applies when the external attribute specifies a name that differs only in case from the declared name. In these situations, the name specified as part of the external attribute is decorated.

For example, given the following declare, the name that the linker sees is `?getenv`.

```
dcl getenv ext('getenv')
   entry( char(*) varz byaddr nonasgn )
   returns( pointer )
   options( nodescriptor linkage(optlink) );
```

Similarly, for the following declare (for the Windows system routine that loads a DLL), the name specified as part of the external attribute is decorated, and the linker sees the name as `_LoadLibraryA@4`.

```
dcl loadlibrarya  ext('LoadLibraryA')
   entry( char(*) varz byaddr nonasgn )
   returns( pointer byvalue )
   options( linkage(stdcall) nodescriptor );
```

If, however, a name is specified as part of the external attribute and that name differs from the declared name by more than its case, then no name decoration occurs.

For example, given the following declare, no name decoration occurs and the name that the linker sees is `?getenv`.

```
dcl getenv ext('?getenv')
 entry( char(*) varz byaddr nonasgn )
 returns( pointer )
 options( nodescriptor linkage(optlink) );
```

Performing name decoration yourself as illustrated in this last example usually makes your code less portable. For instance, only the first declare for `getenv` in the preceding examples is valid for OS/2, Windows, and AIX.

OPTLINK linkage

This is the default calling convention. It is an alternative to SYSTEM linkage that is normally used for calls to the operating system. This linkage provides better total performance than SYSTEM linkage.

Features of OPTLINK

The OPTLINK convention has the following features:

- Parameters are pushed from right to left onto the stack.
- The caller cleans up the stack.
- The general-purpose registers EBX, EDI, and ESI are preserved across the call.
- The general-purpose registers EAX, ECX, and EDX are not preserved across the call.
- Floating-point registers are not preserved across the call.
- The three conforming parameters that are lexically leftmost (conforming parameters are the addresses for all BYADDR parameters and the following BYVALUE parameters: pointer, handle, ordinal, offset, limited entry, real fixed binary, character(1), and nonvarying bits occupying 1 byte or less) are passed in the three unpreserved general-purpose registers.
- Up to four real floating-point or two complex parameters (the lexically first four) are passed in extended precision format (80-bit) in the floating-point register stack.
- All conforming parameters not passed in registers and all nonconforming parameters are passed on the 80386 stack.
- Space for the parameters in registers is allocated on the stack, but the parameters are not copied into that space.
- Conforming return values are returned in EAX.
- Real floating-point return values are returned in extended precision format in the topmost register of the floating-point stack.
- Complex floating-point return values are returned in extended precision format in the topmost two registers of the floating-point stack.
- When you are calling external functions, the floating-point register stack contains only valid parameter registers on entry, and valid return values on exit.
- Functions returning aggregates pass the address of a storage area determined by the caller as a hidden parameter. This area becomes the returned aggregate. The address of this aggregate is returned in EAX.
- The direction flag must be clear upon entry to functions, and clear on exit from functions. The state of the other flags is ignored on entry to a function, and undefined on exit.
- The compiler does not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

General-purpose register implications

Tips for using OPTLINK

By following the tips given below when you use OPTLINK linkage, you can improve the performance of your applications.

- The conforming and floating-point parameters that are most heavily used should be lexically leftmost in the parameter list so they will be considered for registers first. If they are adjacent to each other, the preparation of the parameter list will be faster.
- If you have a parameter that is used near the end of a function, put it at or near the end of the parameter list. If all of your parameters are used near the end of functions, consider using SYSTEM linkage.
- Compile with OPTIMIZE. (See "OPTIMIZE" on page 66.)

General-purpose register implications

Parameters

EAX, EDX, and ECX are used for the lexically first three conforming parameters with EAX containing the first parameter, EDX the second, and ECX the third. Four bytes of stack storage are allocated for each register parameter that is present, but the parameters exist only in the registers at the time of the call.

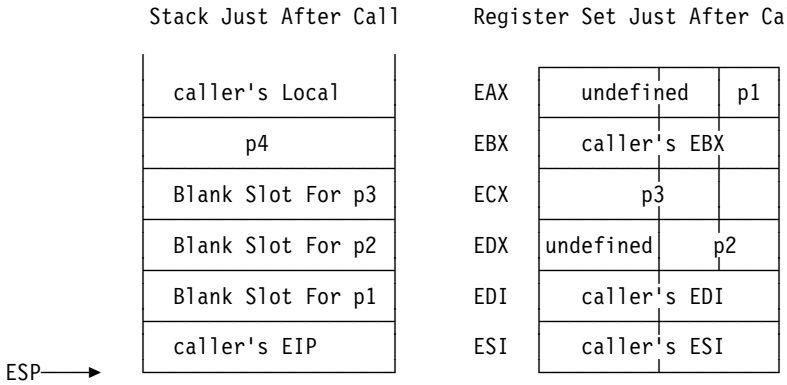
Examples of passing parameters

The following examples are included only for purposes of illustration and clarity and have not been optimized. These examples assume that you are familiar with programming in assembler. In each example, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

Passing conforming parameters to a routine: The following example shows the code sequences and a picture of the stack for a call to the function FUNC1. It is assumed that this program is compiled with the PREFIX(NOFIXEDOVERFLOW) option.

```
dcl func1 entry( char(1),
                 fixed bin(15),
                 fixed bin(31),
                 fixed bin(31) )
    returns( fixed bin(31) )
    options( byvalue nodescriptor );

dcl x fixed bin(15);
dcl y fixed bin(31);

y = func1('A', x, y+x, y);
```

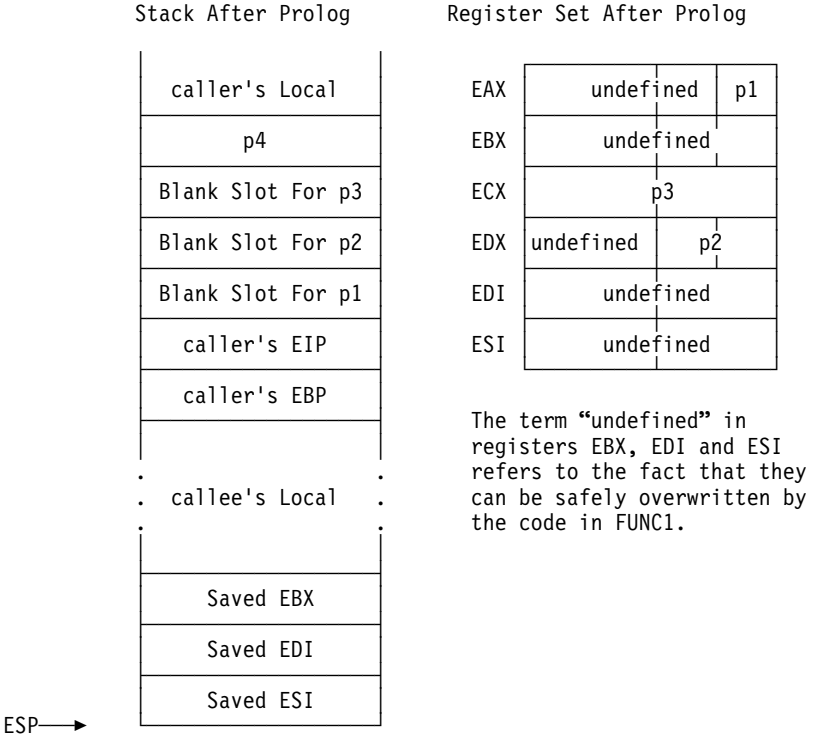
```
caller's Code Up Until Call:

             PUSH     y           ; Push p4 onto the 80386 stack
             SUB      ESP, 12     ; Allocate stack space for
                                  ;  register parameters
             MOV      AL, 'A'     ; Put p1 into AL
             MOV      DX,  x      ; Put p2 into DX
             MOVSX    ECX, DX     ; Sign-extend x to long
             ADD      ECX, y      ; Calculate p3 and put it into ECX
             CALL     FUNC1       ; Make call
```



General-purpose register implications

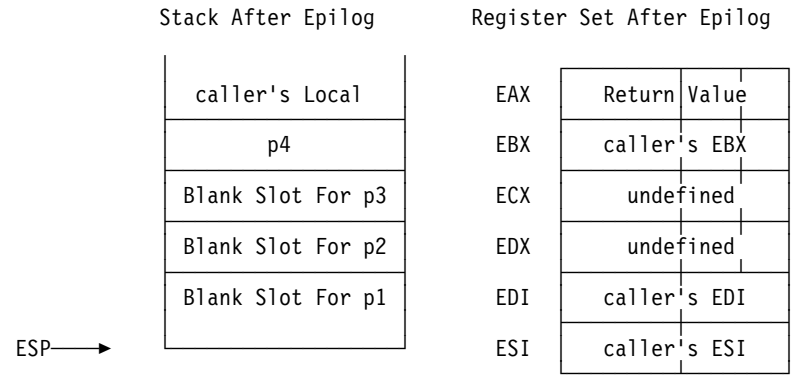
callee's Prolog Code:

```
PUSH     EBP                      ; Save caller's EBP
MOV      EBP, ESP                 ; Set up callee's EBP
SUB      ESP, callee's local size ; Allocate callee's Local
PUSH     EBX                      ; Save preserved registers -
PUSH     EDI                      ;  will optimize to save
PUSH     ESI                      ;  only registers callee uses
```



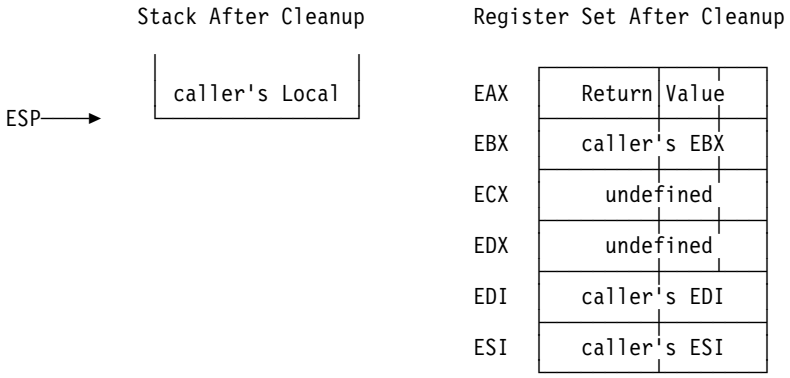
callee's Epilog Code:

```
MOV      EAX, RetVal ; Put return value in EAX
POP      ESI         ; Restore preserved registers
POP      EDI
POP      EBX
MOV      ESP, EBP    ; Deallocate callee's local
POP      EBP         ; Restore caller's EBP
RET                  ; Return to caller
```



caller's Code Just After Call:

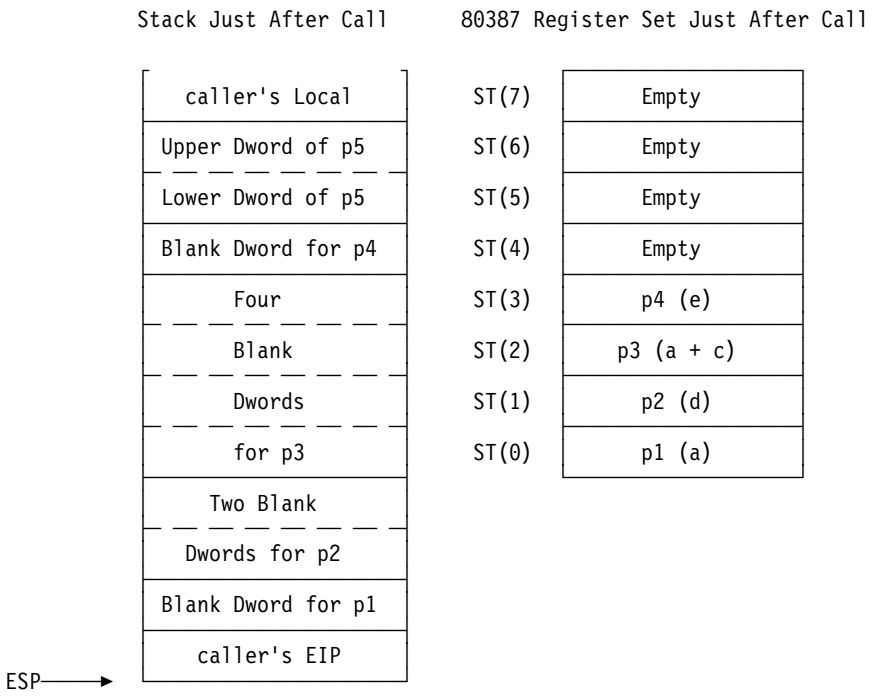
```
ADD     ESP, 16     ; Remove parameters from stack
MOV     y,   EAX    ; Use return value.
```



Passing floating-point parameters to a routine: The following example shows code sequences, 80386 stack layouts, and floating-point register stack states for a call to the routine FUNC2. For simplicity, the general-purpose registers are not shown. It is assumed that this program is compiled with the IMPRECISE option.

General-purpose register implications

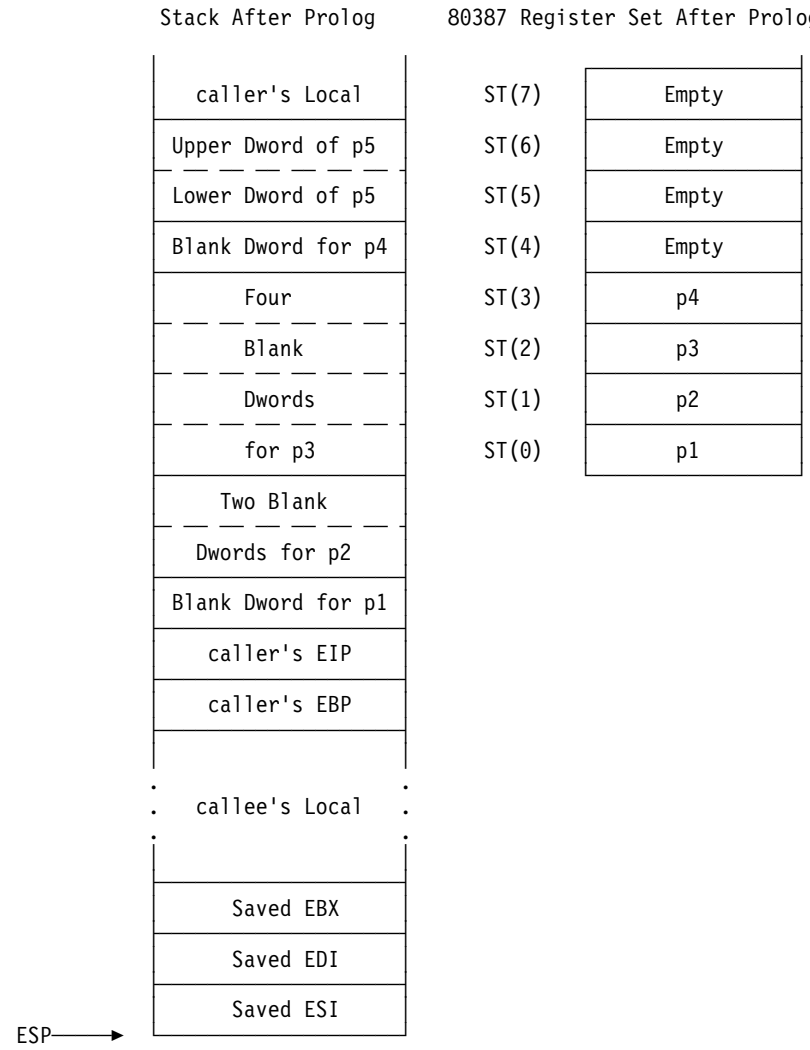
```
 dcl func2 entry( float bin(21),
                  float bin(53),
                  float bin(64),
                  float bin(21),
                  float bin(53) )
     returns( float bin(53) )
     options( byvalue nodescriptor );

 dcl (a, b, c) float bin(53);
 dcl (d, e) float bin(21);

 a = b + func2(a, d, prec(a + c, 53), e, c);

caller's Code Up Until Call:

      PUSH    2ND DWORD OF c       ; Push upper 4 bytes of c onto stack
      PUSH    1ST DWORD OF c       ; Push lower 4 bytes of c onto stack
      FLD     DWORD_PTR e          ; Load e into 80387, promotion
                                   ;  requires no conversion code
      FLD     QWORD_PTR a          ; Load a to calculate p3
      FADD    ST(0), QWORD_PTR c   ; Calculate p3, result is float bin(64)
                                   ;  from nature of 80387 hardware
      FLD     QWORD_PTR d          ; Load d, no conversion necessary
      FLD     QWORD_PTR a          ; Load a, demotion requires conversion
      FSTP    DWORD_PTR [EBP - T1] ; Store to a temp (T1) to convert to float
      FLD     DWORD_PTR [EBP - T1] ; Load converted value from temp (T1)
      SUB     ESP, 32              ; Allocate the stack space for
                                   ;  parameter list
      CALL    FUNC2                ; Make call
```



callee's Prolog Code:

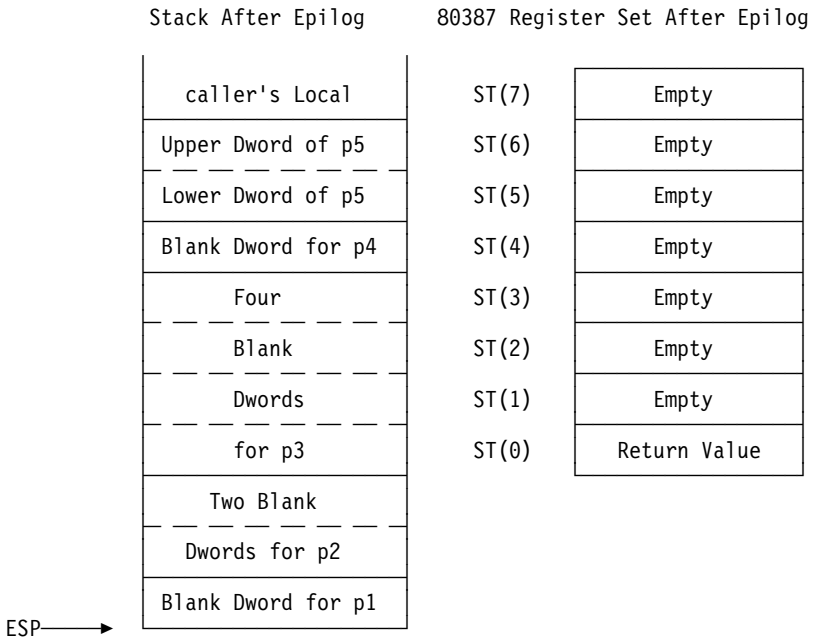
```
PUSH    EBP                     ; Save caller's EBP
MOV     EBP, ESP                ; Set up callee's EBP
SUB     ESP, callee's local size ; Allocate callee's Local
PUSH    EBX                     ; Save preserved registers -
PUSH    EDI                     ;  will optimize to save
PUSH    ESI                     ;  only registers callee uses
```

General-purpose register implications



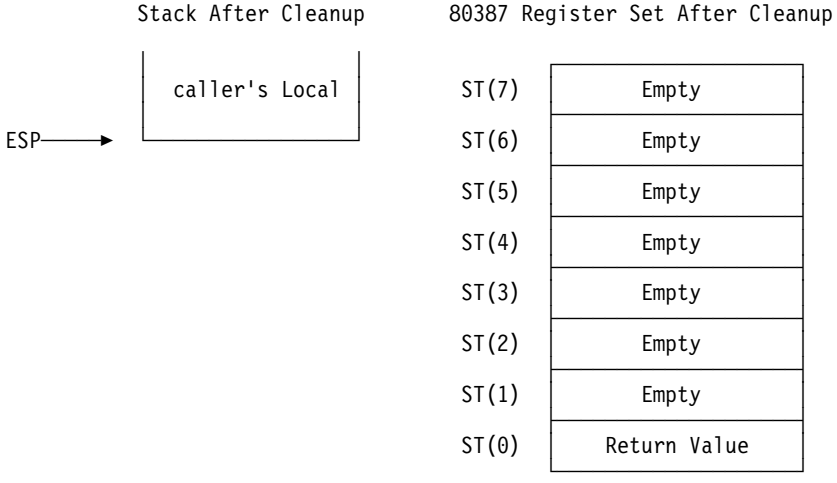
callee's Epilog Code:

```
FLD       RETVAL      ; Load return value onto floating-point stack
POP       ESI         ; Restore preserved registers
POP       EDI
POP       EBX
MOV       ESP, EBP    ; Deallocate callee's local
POP       EBP         ; Restore caller's EBP
RET                   ; Return to caller
```



caller's Code Just After Call:

```
ADD     ESP, 40      ; Remove parameters from stack
FADD    QWORD_PTR b  ; Use return value
FSTP    QWORD_PTR a  ; Store expression to variable a
```



SYSTEM linkage

To use this linkage convention, you must specify the OPTIONS(LINKAGE(SYSTEM)) attribute in the declaration of the function, or specify the DEFAULT(LINKAGE(SYSTEM)) compile-time option.

Features of SYSTEM

The following rules apply to the SYSTEM linkage convention:

- All parameters are passed on the 80386 stack.
- Parameters are pushed onto the stack in right-to-left order.
- The calling function is responsible for removing parameters from the stack.
- All parameters are doubleword (4-byte) aligned.
- Values are returned in the same manner as the OPTLINK linkage.
- The direction flag must be clear upon entry to functions and clear on exit from functions. The state of the other flags is ignored on entry to a function, and undefined on exit.
- The compiler does not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

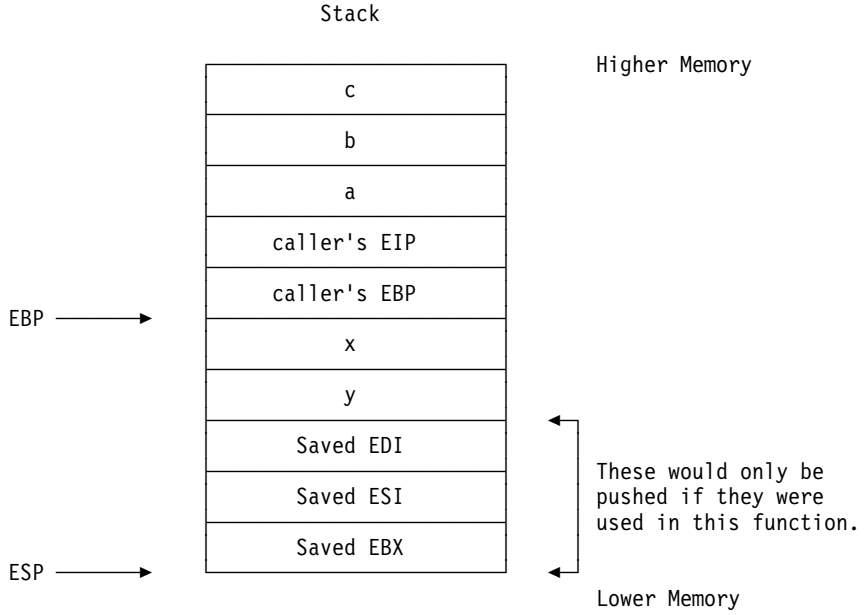
Example using SYSTEM linkage

The following example is included only for purposes of illustration and clarity and has not been optimized. The example assumes that you are familiar with programming in assembler. In the example, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

The following example shows the code sequences and a picture of the stack for a call to the function FUNC3 which has two local variables, x and y (both fixed bin(31)). For the call

```
dcl func3 entry( fixed bin(31),
                 fixed bin(31),
                 fixed bin(31) )
    returns( fixed bin(31) )
    options( byvalue nodescriptor linkage(system) );

 m = func3(a,b,c);
```

the stack for the call to FUNC3 would look like this:



The instructions used to build this activation record on the stack look like this on the calling side:

```
PUSH    c
PUSH    b
PUSH    a
MOV     AL, 3H
CALL    func3
    .
    .
ADD     ESP, 12    ; Cleaning up the parameters
    .
    .
MOV     m, EAX
    .
    .
```

STDCALL linkage

For the callee, the code looks like this:

```
func3 PROC
      PUSH    EBP
      MOV     EBP, ESP           ;  Allocating 8 bytes of storage
      SUB     ESP, 8             ;   for two local variables.
      PUSH    EDI                ; These would only be
      PUSH    ESI                ;  pushed if they were used
      PUSH    EBX                ;  in this function.
      .
      .
      MOV     EAX, [EBP - 8]    ; Load y into EAX
      MOV     EBX, [EBP + 12]   ; Load b into EBX
      .
      .
      XOR     EAX, EAX           ; Zero the return value
      POP     EBX                ; Restore the saved registers
      POP     ESI
      POP     EDI
      LEAVE                      ; Equivalent to  MOV   ESP, EBP
                                 ;                POP   EBP
      RET
func3 ENDP
```

The saved register set is EBX, ESI, and EDI. The other registers (EAX, ECX, and EDX) can have their contents changed by a called routine.

Under some circumstances, the compiler does not use EBP to access automatic and parameter values, thus increasing the application's efficiency. Whether it is used or not, EBP does not change across the call.

When passing aggregates by value, the compiler generates code to copy the aggregate on to the 80386 stack. If the size of the aggregate is larger than an 80386 page size (4K), the compiler generates code to copy the aggregate backward (that is, the last byte in the aggregate is the first to be copied). This operation ensures that the OS/2 guard page method of stack growth functions properly in the presence of large aggregates being passed by value.

Aggregates are not returned on the stack. The caller pushes the address where the returned aggregate is to be placed as a lexically first hidden parameter. A function that returns an aggregate must be aware that all parameters are 4 bytes farther away from EBP than they would be if no aggregate return were involved. The address of the returned aggregate is returned in EAX.

STDCALL linkage (Windows only)

To use this linkage convention, you must specify the OPTIONS(LINKAGE(STDCALL)) attribute in the declaration of the function, or specify the DEFAULT(LINKAGE(STDCALL)) compile-time option.

Features of STDCALL

The following rules apply to the STDCALL calling convention:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The *called* function removes the parameters from the stack.
- Floating point values are returned in `ST(0)`, the top register of the floating point register stack. Functions returning aggregate values return them as follows:

Size of Aggregate	Value Returned in
8 bytes	EAX-EDX pair
5, 6, 7 bytes	EAX The address to place the return values is passed as a hidden parameter in EAX.
4 bytes	EAX
3 bytes	EAX The address to place the return values is passed as a hidden parameter to EAX.
2 bytes	AX
1 byte	AL

  For functions that return aggregates 5, 6, 7 or more than 8 bytes in size, the address to place the return values is passed as a hidden parameter, and the address is passed back in `EAX`.
- STDCALL has the restriction that an unprototyped STDCALL function with a variable number of arguments will not work.
- Function names are decorated with an underscore prefix, and a suffix which consists of an at sign (@), followed by the number of bytes of parameters (in decimal). Parameters of less than four bytes are rounded up to four bytes. Structure sizes are also rounded up to a multiple of four bytes. For example, consider a function `fred` prototyped as follows:

  ```
  dcl fred ext entry (fixed bin(31) byvalue, fixed bin(31) byvalue, fixed bin(15) byvalue);
  ```

  It would appear as follows in the object module:

  ```
  _FRED@12
  ```

When building export lists in `.DEF` files, the decorated version of the name should be used. If you use undecorated names in the DEF file, you must give the object files to ILIB along with the DEF file. ILIB uses the object files to determine how each name ended up after decoration.

Examples using the STDCALL convention

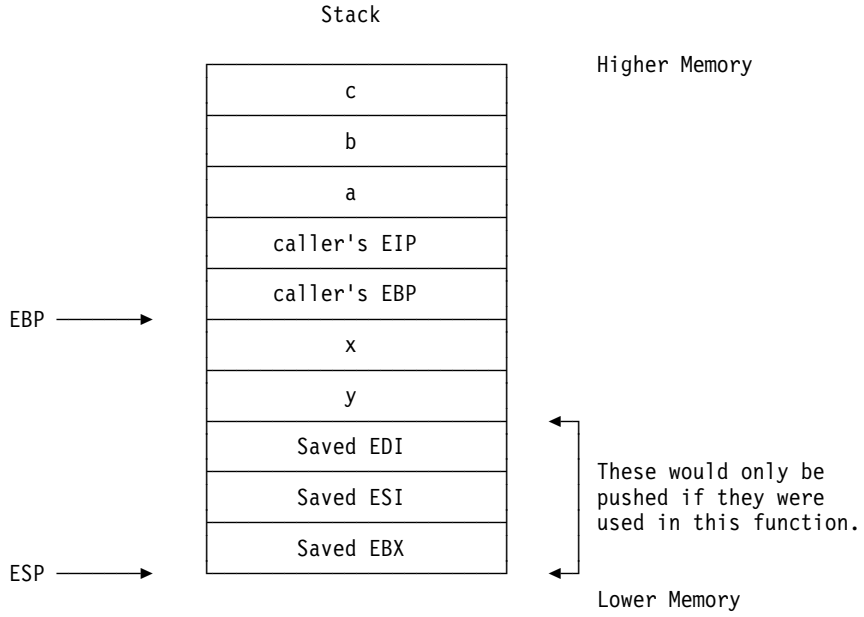
The following examples are included for purposes of illustration and clarity only. The examples assume that you are familiar with programming in assembler. In the examples, the stack grows toward the bottom of the page, and `ESP` always points to the top of the stack.

STDCALL linkage

For the following call, `a, b,` and `c` are 32-bit integers and `func` has two local variables, `x` and `y` (both 32-bit integers):

```
m = func(a,b,c)
```

The stack for the call to FUNC would look like this:



The instructions used to create this activation record on the stack look like this on the calling side:

```
PUSH c
PUSH b
PUSH a
CALL _func@12
  .
  .
MOV  m, EAX
  .
  .
```

For the callee, the code looks like this:

```
_func@12 PROC
         PUSH    EBP
         MOV     EBP, ESP           ; Allocating 8 bytes of storage
         SUB     ESP, 8             ;  for two local variables.
         PUSH    EDI                ; These would only be
         PUSH    ESI                ;  pushed if they were used
         PUSH    EBX                ;  in this function.
         .
         .
         MOV     EAX, [EBP - 8]    ; Load y into EAX
         MOV     EBX, [EBP + 12]   ; Load b into EBX
         .
         .
         XOR     EAX, EAX           ; Zero the return value
         POP     EBX                ; Restore the saved registers
         POP     ESI
         POP     EDI
         LEAVE                      ; Equivalent to  MOV   ESP, EBP
                                    ;                POP   EBP
         RET     0CH
_func@12 ENDP
```

The saved register set is EBX, ESI, and EDI.

Structures are not returned on the stack. The caller pushes the address where the returned structure is to be placed as a lexically first hidden parameter. A function that returns a structure must be aware that all parameters are four bytes farther away from EBP than they would be if no structure were involved. The address of the returned structure is returned in EAX.

Using WinMain (Windows only)

WIN You can use WinMain by specifying OPTIONS(WINMAIN) on the procedure statement (see the PL/I language reference information for syntax). This automatically implies LINKAGE(STDCALL) and EXT('WinMain').

Your WinMain routine needs four parameters:

- An instance handle
- A previous handle
- A pointer to the command line
- An integer to be passed to ShowWindow

These are the same four parameters expected by WinMain in C. The calls made inside this routine are the same as those expected from a C routine.

An example `guisamp.pli` is provided in the samples directory (see the program prolog for more details).

CDECL linkage

To use this linkage convention, you must specify the OPTIONS(LINKAGE(CDECL)) attribute in the declaration of the function, or specify the DEFAULT(LINKAGE(CDECL)) compile-time option.

Features of CDECL

The following rules apply to the CDECL calling convention:

- All parameters are passed on the stack.
- The parameters are pushed onto the stack in a lexical right-to-left order.
- The *calling* function removes the parameters from the stack.
- Floating point values are returned in `ST(0)`. All functions returning non-floating point values return them in `EAX`, except for the special case of returning aggregates less than or equal to eight bytes in size. For functions that return aggregates less than or equal to four bytes in size, the values are returned as follows:

Size of Aggregate	Value Returned in
8 bytes	EAX-EDX pair
5, 6, 7 bytes	EAX The address to place return values is passed as a hidden parameter in EAX.
4 bytes	EAX
3 bytes	EAX The address to place return values is passed as a hidden parameter to EAX.
2 bytes	AX
1 byte	AL

  For functions that return aggregates 5, 6, 7 or more than 8 bytes in size, the address to place the return values is passed as a hidden parameter, and the address is passed back in `EAX`.
- Function names are decorated with an underscore prefix when they appear in object modules. For example, a function named `fred` in the source program will appear as `_fred` in the object.

When building export or import lists in `.DEF` files, the decorated version of the name should be used. If you used undecorated names in the DEF file, you must give the object files to ILIB along with the DEF file. ILIB uses the object files to determine how each name ended up after decoration.

Examples using the CDECL convention

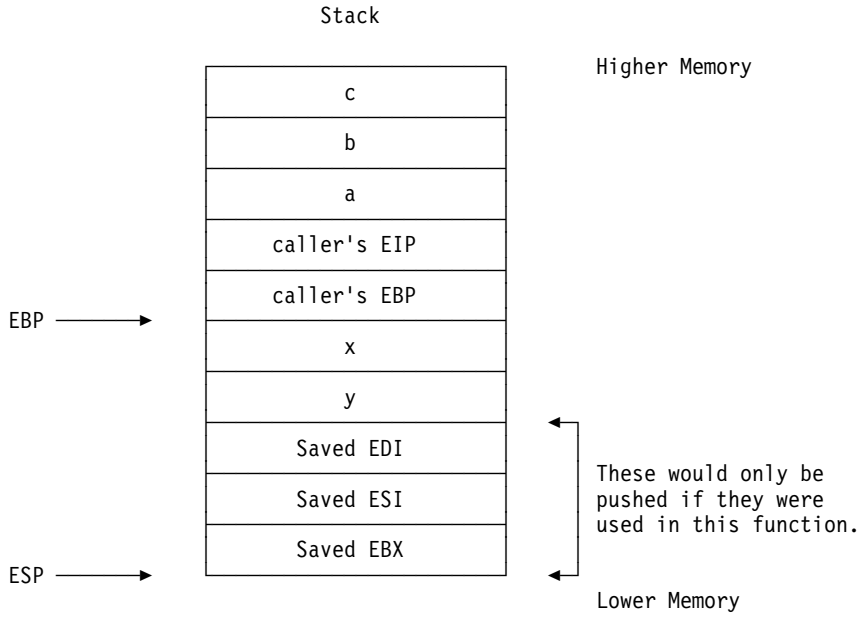
The following examples are included for purposes of illustration and clarity only. They have not been optimized. The examples assume that you are familiar with programming in assembler. In the examples, the stack grows toward the bottom of the page, and `ESP` always points to the top of the stack.

Consider the following call:

```
m = func(a,b,c);
```

The variables `a`, `b`, and `c` are 32-bit integers and FUNC has two local variables, `x` and `y` (both 32-bit integers).

The stack for the call to FUNC would look like this:



The instructions used to create this activation record on the stack look like this on the calling side:

```
PUSH c
PUSH b
PUSH a
CALL _func
  .
  .
ADD  ESP, 12   : cleaning up the parameters
  .
  .
MOV  m, EAX
  .
  .
```

CDECL linkage

For the callee, the code looks like this:

```
_func PROC
      PUSH    EBP
      MOV     EBP, ESP           ; Allocating 8 bytes of storage
      SUB     ESP, 08H           ;  for two local variables.
      PUSH    EDI                ; These would only be
      PUSH    ESI                ;  pushed if they were used
      PUSH    EBX                ;  in this function.
      .
      .
      MOV     EAX, [EBP - 8]    ; Load y into EAX
      MOV     EBX, [EBP + 12]   ; Load b into EBX
      .
      .
      XOR     EAX, EAX           ; Zero the return value
      POP     EBX                ; Restore the saved registers
      POP     ESI
      POP     EDI
      LEAVE                      ; Equivalent to  MOV   ESP, EBP
                                 ;                POP   EBP
      RET
_func ENDP
```

The saved register set is EBX, ESI, and EDI. In the case where the structure is passed as a value parameter and the size of the structure is 5, 6, 7, or more than 8 bytes in size, the address to place the return values is passed as a hidden parameter, and the address passed back in EAX.

Chapter 25. Using PL/I in mixed-language applications

Matching data and linkages

Within the workstation environment, there are occasions when you want to develop mixed-language applications with PL/I being one of the languages involved. For example, an application could be constructed with the main program written in C and a dynamic link library (DLL) written in PL/I. Another possibility is an application using OS/2 REXX which can load and call PL/I routines packaged in a PL/I DLL.

Perhaps you want to construct an application using software from an outside vendor. Using a vendor's prepackaged program, you can supply a user exit in the form of a DLL written in PL/I.

Creating mixed-language applications is generally challenging and you have to consider many factors that do not exist when coding in a single language. Typically, high level programming languages from different vendors (for example, C, C++, COBOL, and PL/I) require the use of specific run-time environments as implemented by the run-time libraries of the distinct languages. Areas in which these languages might not work well together include:

- Implementations and usages of data types
- Data alignments
- Exception handling facilities
- Run-time environment initialization and termination
- User exit routines
- Input and output facilities

These inconsistencies in behavior can cause unexpected run-time behavior that can arise in some mixed-language program execution scenarios.

Matching data and linkages

For any routine to invoke another routine successfully, the two routines should have matching views of shared interfaces. When one of the routines is not coded in PL/I, these interfaces are limited by

- What data is passed
- How data is passed
- Where data is passed

The sections that follow describe these situations in more detail. Mismatched views of shared interfaces is a common problem in mixed language applications. Important points to remember are:

- Arguments and parameters must match
- Data that is meant to be received by value should be passed by value
- Both the called and calling routines should use the same linkage.

What data is passed

PL/I and C routines 'communicate' by passing and returning data of equivalent data types. PL/I and non-PL/I routines should **not** communicate by using external static variables. Table 33 on page 443 lists the scalar data types which are equivalent between PL/I and C.

Table 33. Equivalent data types between C and PL/I

C Data Type	PL/I Data Type
signed char	FIXED BIN(7,0)
unsigned char	UNSIGNED FIXED BIN(8,0) or CHAR(1)
signed short	FIXED BIN(15,0)
unsigned short	UNSIGNED FIXED BIN(16,0)
signed (long) int	FIXED BIN(31,0)
unsigned (long) int	UNSIGNED FIXED BIN(31,0)
float	FLOAT BIN(21) FLOAT DEC(6)
double	FLOAT BIN(53) FLOAT DEC(16)
long double	FLOAT BIN(64) FLOAT DEC(18)
enum	ORDINAL
<non-function-type> *	POINTER or HANDLE
<function-type> *	ENTRY LIMITED

As is illustrated in the last row of the table, a C function pointer is not equivalent to a PL/I entry variable unless the entry variable is LIMITED. Errors caused by this mistake are hard to detect.

Arrays of equivalent types are equivalent as long as they have the same number of dimensions and the same lower and upper bounds. In C, you cannot specify lower bounds, and the actual upper bound is one less than the number you specify. For example, consider this array declared in C:

```
short x [ 6 ];
```

In PL/I, the array would be declared as follows:

```
dcl x(0:5) fixed bin(15);
```

Structures and unions of equivalent types are also equivalent if their elements are mapped to the same offsets. The offsets are the same if there is no padding between elements. If the elements of a structure (or union) are all UNALIGNED, PL/I does not use padding. When some elements are ALIGNED, you can determine if there is any padding by examining the AGGREGATE listing. PL/I regards strings as scalars but C does not; therefore, none of the previous discussion applies to strings.

C bit fields have only nominal resemblance to PL/I bit strings:

- C bit fields are limited to 32 bits, while PL/I bit strings can be as long as 32767 bits
- C bit fields are not always mapped in left-to-right order. Some Intel C compilers would map the following C structure so that it is equivalent to the PL/I structure:

  C Structure

Matching data and linkages

```
struct { unsigned byte1 :8;
         unsigned byte2 :8;
         unsigned byte3 :8;
         unsigned byte4 :8;
       } bytes;
```

PL/I Structure

```
dcl
  1 bytes,
    2 byte1 bit(8),
    2 byte2 bit(8),
    2 byte3 bit(8),
    2 byte4 bit(8);
```

Other C compilers would map the original structure with the bytes reversed so that it would be equivalent to this PL/I structure.

PL/I Structure

```
dcl
  1 bytes,
    2 byte4 bit(8),
    2 byte3 bit(8),
    2 byte2 bit(8),
    2 byte1 bit(8);
```

Strictly speaking, C has no character strings, but only pointers to char. However, by common usage, a C string is a sequence of characters the last of which has the value X'00'. Thus, in the example below, *address* is a C 'string' that could hold up to 30 non-null characters.

```
char address [ 31 ];
```

The following PL/I declare most closely resembles the C 'string'.

```
dcl address char(30) varyingz;
```

In the declarations of C functions, strings are usually declared as char*. For example, the C library function *strcspn* could be declared as:

```
int strcspn( char * string1, char * string2 );
```

The PL/I declare for the same function would be:

```
dcl strcspn entry( char(*) varyingz,
                   char(*) varyingz )
            returns( fixed bin(31) );
```

In the preceding examples, both the C and PL/I declarations are incomplete. Complete versions are given and explained later in this chapter.

How data is passed

Both PL/I and C support various methods of passing data. To understand these methods, you must know the following terms:

Parameter
A variable declared in a PL/I procedure or function definition. For example, *seed* is a parameter in the following PL/I function definition.

```
funky:
   proc( seed )
   returns( fixed bin(31) );

   dcl seed fixed bin(31);
   ⋮
end funky;
```

Argument
A variable or value actually passed to a routine. When the function *funky* (from the preceding example) is invoked by `rc = funky( seed );`, *seed* is an argument.

By value
The value of the argument is passed. When a calling routine passes an argument by value, the called routine **cannot** alter the original argument.

By address
The address of the argument is passed. When a calling routine passes an argument by address, the called routine **can** alter the caller's argument.

C passes all parameters by value, but PL/I (by default) passes parameters by address. PL/I also supports passing parameters by value except for arrays, structures, unions, and strings with length declared as *.

As is described in more detail in the *PL/I Language Reference*, you can indicate if a parameter is passed by address or by value by declaring it with the BYADDR or BYVALUE attribute. In the following example, the first parameter to *modf* is passed by value, while the second is passed by address.

```
dcl modf entry( float bin(53) byvalue,
                float bin(53) byaddr )
         returns( float bin(53) );
```

The corresponding C declaration is:

```
double modf( double x, double * intptr );
```

If the BYADDR or BYVALUE attributes are not explicit in the declaration, you can specify them in the options list for that entry. The following declare uses the options list making it equivalent to the previous example.

```
dcl modf entry( float bin(53),
                float bin(53) byaddr )
         returns( float bin(53) )
         options( byvalue );
```

Matching data and linkages

Even when a parameter is passed by address, its value might not be changed by the receiving routine. You can indicate this in PL/I by adding the attribute NONASSIGNABLE (or NONASGN) to the declaration for that parameter. The following partial declaration indicates that neither of the arguments to the function *strcspn* is altered by that function:

```
dcl strcspn entry( nonasgn char(*) varyingz,
                   nonasgn char(*) varyingz )
            returns( fixed bin(31) );
```

The corresponding C declaration is:

```
int strcspn( const char * string1, const char * string2 );
```

A routine must agree with any routines that call it about how data is passed between them. You can avoid potential problems by giving the compiler enough information to detect these kinds of mismatches. For example, while the following declare is technically equivalent to the declare for *modf* in the sample code shown earlier, it allows the address of any argument to be passed as the second argument. The earlier declares would require the second argument to have the correct type.

```
dcl modf entry( float bin(53),
                pointer )
         returns( float bin(53) )
         options( byvalue );
```

Finally, when PL/I passes some data types (strings, arrays, structures, and unions), it also, by default, passes a *descriptor* that describes data extents (maximum string length, array bounds, etc.). Since C routines cannot consume PL/I descriptors, you should keep descriptors from being passed between C and PL/I routines. You can do this by adding the NODESCRIPTOR option to the OPTIONS attribute in the declaration for the C entry, for example:

```
dcl strcspn entry( nonasgn byaddr char(*) varyingz,
                   nonasgn byaddr char(*) varyingz )
            returns( fixed bin(31) )
            options( nodescriptor );
```

Where data is passed

It is as important for interacting routines to agree on what and where data is passed as it is for them to agree on how data is passed. With both PL/I and C, data can be passed on the stack, in general registers, or in floating-point registers.

In PL/I, the LINKAGE option (in the OPTIONS option of the procedure statement and entry declaration) determines where data is passed. One common way that errors in data location occur is if you specify mismatched linkage types (or fail to specify a linkage type when the default is incorrect).

VisualAge PL/I for OS/2 supports three 32-bit linkage types— OPTLINK, CDECL and SYSTEM. VisualAge PL/I for Windows also supports three 32-bit linkage types— OPTLINK, CDECL and STDCALL. The following PL/I declaration indicates that the function *dosSleep* uses the SYSTEM linkage:

```
dcl dosSleep entry( fixed bin(31) byvalue )
             returns( fixed bin(31) )
             options( linkage(system) );
```

The options list should specify the linkage used by any C routines you call. While both the PL/I and VisualAge for C++ compilers use OPTLINK as their default linkage, many C routines on OS/2 use the SYSTEM linkage. For such routines, LINKAGE(SYSTEM) should be specified in the OPTIONS attribute. Many C routines on Windows use the STDCALL linkage, and for these routines, LINKAGE(STDCALL) should be specified in the OPTIONS attribute. For instance, you would declare the Windows equivalent of DosSleep as:

```
dcl Sleep    ext('Sleep')
             entry( fixed bin(31) byvalue )
             returns( fixed bin(31) )
             options( linkage(stdcall) );
```

Maintaining your environment

In order for PL/I (and many other languages) to work correctly, you must not damage the runtime environment they establish. When interlanguage calls are involved, this means that:

- Any routine that registers an exception handler should deregister that handler before returning to PL/I.
- Out-of-block GOTOs are permitted only if the soure and target blocks are coded in the same language and any intervening blocks are coded in the same language.

Invoking non-PL/I routines from a PL/I MAIN

If your main routine is coded in PL/I, you can call two kinds of non-PL/I routines:

- System routines (such as DOS and PM services on OS/2)
- C, COBOL, or REXX routines

System routines do not require their own run-time environment, and they can be linked directly into a PL/I executable (.EXE) file or dynamic link library (.DLL). With the exception of IBM VisualAge C/C++ routines, all other non-PL/I routines should **not** be linked directly into an .EXE or .DLL. They should be linked instead into a .DLL so that any run-time environment initialization that they require can be performed when that .DLL is loaded.

IBM VisualAge C/C++ routines can be linked with PL/I. However, if C routines are linked with PL/I and any of them use C library functions (or are C library functions themselves), the C runtime must be initialized before any routines are called. The C runtime can be initialized by calling the following routine

```
dcl _CRT_init  ext('_CRT_init')
               entry()
               returns( optional fixed bin(31) )
               options( linkage(optlink) );
```

Also, in order to ensure that the C runtime closes all files it opened and returns any other system resources it may have acquired, you have to terminate the C runtime by calling

```
dcl _CRT_term  ext('_CRT_term')
               entry()
               returns( optional fixed bin(31) )
               options( linkage(optlink) );
```

Invoking PL/I routines from a non-PL/I main

The PL/I run-time environment has the ability to:

- Self-initialize when a PL/I DLL is dynamcally loaded from a non-PL/I main program.
- Exist with a non-PL/I language run-time environment with minimal conflicts.

Any PL/I routine called directly from non-PL/I routines must have the FROMALIEN option in the OPTIONS option and must not specify the MAIN option.

A PL/I routine invoked from a non-PL/I routine should handle any exceptions that occur in PL/I code and returns to the non-PL/I using a RETURN or END statement in the first PL/I procedure (see “Using ON ANYCONDITION”)

The PL/I run-time implicitly frees any resources acquired by PL/I, but not until the application terminates.

You can also explicitly resources through various PL/I statements:

- RELEASE * - releases all fetched modules
- FLUSH FILE(*) - flushes all file buffers
- CLOSE FILE(*) - closes all open files

Using ON ANYCONDITION

Any application should be able to handle all exceptions that occur within it and return 'normal' control to the calling program. PL/I exception-handling facilities and ANYCONDITION ON-units help make this possible.

The first executable statement in any PL/I routine that is called from a non-PL/I routine should be an ON ANYCONDITION statement. This statement should contain code to handle any condition not handled explicitly by other ON-units. If a condition arises that cannot be handled, use a GOTO statement pointing to the last statement that would normally be executed in the routine, for example:

```
pliapp:
  proc( p1, ...,  pn )
  returns( ... )
  options( fromalien );

/* declarations of paramaters, if any */

/* declarations of other variables    */

on anycondition
  begin;
    /* handle condition if possible   */

    /* if unhandled, set return value */
    goto return_stmt;
  end;

/* mainline code */

return_Stmt:
return( ... );

end_stmt:
end pliapp;
```

For PL/I routines that are not functions, the target for the GOTO should be the END statement in the routine.

Creating mixed-language applications

Several sample programs have been included with the product to help you understand how to design mixed-language applications. These samples, including source files and module definition files, are packaged in the \SAMPLES directory.

This section describe specific coding considerations for Borland C and REXX/2 when used in mixed applications with PL/I.

Borland C/C++ for OS/2

OS/2 Observe the following guidelines when using Borland C/C++ in mixed-language applications:

- Borland C/C++ supports only SYSTEM, and not OPTLINK, linkage. Since the PL/I default is OPTLINK, you must make sure that PL/I routines that exchange parameters with Borland C/C++ routines are compiled with SYSTEM linkage.
- With the Borland compiler, unless the *-a* compiler option is used, only 10 bytes are used for long doubles. If the *-a* option is used, 12 bytes are used. PL/I uses 16 bytes for the equivalent data type. Because of this difference, you have to add dummy fields after long doubles in C structures that are used by PL/I.

- When you specify the *-a* compile-time option, 4 bytes are used for signed and unsigned shorts. Since PL/I uses 2 bytes for the equivalent data types, you have to add dummy fields after shorts in C structures compiled with this option.
- Because of the different storage sizes for LONG DOUBLEs and SHORTs, these types should be passed by-reference when passed as scalar arguments between C and PL/I.
- Arrays of LONG DOUBLEs and SHORTs should not be used as arguments between the two languages.

REXX/2

OS/2 Observe the following guidelines when using REXX in mixed-language applications:

- If a PL/I routine is called from REXX, the routine should be coded as a function returning a FIXED BIN(31) value. If the PL/I routine is not coded as a function, the behavior of the REXX code after the call is unpredictable although a REXX error code of 40 often results.
- When trying to run a mixed REXX and PL/I program from multiple sessions, the PL/I DLL must be built with the attributes DATA MULTIPLE NONSHARED in its definition (.DEF) file or a protection exception occurs in the second session.
- The RxFuncDrop command in REXX does NOT release DLLs; it only releases the function. You have to exit out of the session to free the DLL.
- A PL/I procedure that fetches a REXX DLL such as REXXUTIL.DLL should not use the PL/I RELEASE statement explicitly to free the DLL. An explicit release causes the run-time message IBM0596 and ONCODE = 9256 to be issued because REXX DLL is being shared across the system and PL/I does not have authority free it.

Chapter 26. Calling PL/I native methods from Java

This chapter gives an overview of Java and explains why you might be interested in using it with PL/I. Because the actual implementation of calling PL/I from Java is subject to frequent change, these steps are described in a file named `plijava.htm` shipped with VisualAge PL/I. You need access to a browser to view the information in this file.

OS/2 For OS/2, the browsable file is in the `d:\ibmpli\help` directory and the sample programs are in `d:\ibmpli\samples`, assuming you used `d:\ibmpli\` to install the product.

WIN For Windows NT, the browsable file is in the `d:\ibmpliw\books` directory and the sample programs are in `d:\ibmpliw\samples`, assuming you used `d:\ibmpliw\` to install the product.

What is java?

Java is an object-oriented programming language invented by Sun Microsystems and provides the most powerful way to make Internet documents interactive. Java was derived from C++, but is much more approachable. Java designers simplified it by removing memory management, pointers, and operator overloading. They also made Java more 'internet aware' by adding classes and methods for networking (TCP/IP, URLs, Sockets) and Applets.

The Java Development Kit 1.1.2 (most current version as of the printing of the document) has classes that support the Abstract Window Toolkit (AWT), I/O, utilities, and many other functions.

The Java compiler does not produce platform dependent machine code. Instead, it produces what are known as *java byte codes* in a Java `.class` file. These byte codes are not specific to any one processor or platform.

Java `.class` files are executed by an interpreter known as a Java virtual machine. The `.class` files are machine independent, meaning they can be compiled on one platform and run on any other platform that has a Java virtual machine. This is a great benefit to programmers who want to write a program once and then have it run on many platforms without altering the code.

Because Java is very new, it is still evolving. Java support on OS/2 first came out at the Java 1.0.2 level. The current support on OS/2 is for Java 1.1.1 which is generally available at the time this information was written. Sun Microsystems provides a Software Development Kit (SDK) that supports Java 1.1.3 on Windows platforms. There are significant changes between JDK 1.0.2 and JDK 1.1.3. **You should always start with the newest version of Java available when developing Java applications.**

The Java Development Kit

The Java Development Kits come with a Java compiler (javac), a Java runtime, debugger and assorted tools. The PL/I sample program provide with VisualAge PL/I was developed using the OS/2 Java 1.1.1 and Sun Microsystems Windows Java Development Kits 1.1.2 and 1.1.3. These Java Development Kits are free.

Places to download the Java Development kits for free:

IBM Java Development Kit 1.1 for OS/2:
`http://service.boulder.ibm.com/asd-bin/doc/en_us/java111/Heat.htm`

Sun Microsystems Java Development Kit 1.1.3 for Windows:
`http://java.sun.com/products/jdk/1.1/.htm`

Java programs have a suffix of `.java`. The Java compiler is case sensitive and requires that the source file name matches the class name it contains. For example, the class 'MyFirstJavaClass { }' is different than 'Myfirstjavaclass { }'. The source file for MyFirstJavaClass must be named `MyFirstJavaClass.java`. The compiler names the output file `MyFirstJavaClass.class`.

If you have more than one Java class in your source file, the Java compiler creates a separate `.class` file for each of the classes it finds.

Java applications and applets

A Java program can be either an application or an applet.

Java applications are self contained programs much like traditional PL/I programs. They contain one or more classes, few or many methods, and run on their own.

Java applets are designed to run in a browser environment. This makes them a good choice for internet or client/server-based applications. Applets operate with the following restrictions:

- Applets cannot read or write to your file system, unless directories are specifically set aside for that purpose.
- Applets cannot usually communicate with a server other than the one that originally sent the applet.
- Applets cannot run any programs on your system.
- Applets cannot load programs native to your system, including shared libraries or DLLs.

These restrictions are intended to protect your system from a 'misbehaving' applet.

Applets and applications also differ in their structure. Here are a couple of the more significant differences:

- Applets inherit from the `java.applet.Applet` class while applications inherit from the `java.awt.Frame` class.

- Applet execution begins with an 'init()' method while applications begin with a constructor.

It is possible to create a Java program that can be run either as an applet or an application as long as it adheres to the applet restrictions.

You can run a Java application by entering a command like the one shown here:

```
java myApplication
```

The `java` command invokes the Java virtual machine and passes it the name of your Java program. This command is carried out based on the assumption that a file named `myApplication.class` exists somewhere on the CLASSPATH.

You run a Java applet by using one of the following:

- A Java enabled browser
- The *appletviewer* command (Windows or Unix)
- The *javapm* command (OS/2)

For example, the following command would run your `myApplet` applet on OS/2 by invoking the java interpreter and running the file `myApplet.class`.

```
javapm myApplet
```

At the time this information is being written, Java Version 1.1 is so new that no browser currently supports it. Until one does, you can use javapm and appletviewer to mimic the browser environment.

There are many other issues and more information necessary to create and run Java programs. If you are interested in learning more about programming in Java, you might want to invest in a good Java book or set of online information.

Java native methods, what are they and why use them?

Java is a fairly complete programming language, however, there could be situations in which you want to call a program written in another programming language. In Java, you do this with a method call to a native language, known as a *native method*.

Why call a native language?

Here are some possible reasons why you might want to call a native language:

- The native language has a special capability you need or one that Java lacks.

  For example, you might need to access a specific operating system feature or a piece of special hardware.

- You need speed.

  You could have an intensive series of calculations that need to be performed.

- Your skill set is broader in the native language.

  If you have skilled PL/I programmers, you want to get the most out of them.

- You have made a significant investment in code.

  Rather than move to a new language, it is in your best interest to keep using the code that has been valuable to you for years.

To be fair, here are some reasons why you might not want to call a native language:

- The availability of your product is compromised.

  For example, if your program calls a native method in a programming language that runs only on UNIX, you will not be able to move it to another platform.

- The portability of your product is compromised.

  Even if the programming language you choose is widely used, you still need to recompile your native program on each platform for which you deliver.

  Fortunately, this is not such a problem with PL/I. You can use the PL/I macro facility to write your program using conditional code. This allows you to compile correctly on your target operating system, as illustrated by the Java PL/I sample application.

  Portability is not a problem with the Java bytes portion of your product as it can be run 'as is' on any supported platform.

- If your product is an applet, it cannot bring native code to the client.

  For example, an applet is restricted from executing native shared or load libraries on a client machine. This shortcoming can be overcome by designing your application to use native methods in the server part of the code, assuming a client/server environment.

Benefits of calling a native language

An important benefit of using a native language is being able to leverage your current PL/I skills and PL/I program base. You can continue to use the highly-trained PL/I programmers in your shop to produce powerful applications as well as reuse your existing code.

You can develop a Graphical User Interface (GUI) using Java that will run on all of the supported platforms that call your native language. By combining a portable GUI and a set of powerful native methods, you are able to move your application from platform to platform.

As you spend time learning about Java through the sample application provided, you will learn how to connect your legacy code into the flexible world of the internet. As was mentioned in the introduction to this chapter, the methods used to create this sample application scenario are so dynamic that the steps are provided in an online document so we can provide updates as technology evolves.

Chapter 27. Using sort routines

Comparing sort programs

VisualAge PL/I supports the PLISRT*x* (*x* = A, B, C, or D) built-in subroutines. To use the PLISRT*x* subroutines, you need to:

- Include a call to one of the subroutines and pass it the information on the fields to be sorted. This information includes the length of the records, the name of a variable to be used as a return code, and other information required to carry out the sort.
- Specify the data sets required by the sort program in DD statements.

Windows Users

The PLISRTx routines are supported on Windows. In order to use them, however, you must have the SMARTsort for Windows product installed (separately orderable).

When used from PL/I, these subroutines sort records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted can be either a data set or a PL/I procedure written by the programmer that the sort program calls each time a record is required for the sort. Similarly, the destination of the sort can be a data set or a PL/I procedure that handles the sorted records.

Comparing S/390 and workstation sort programs

If your existing mainframe programs contain CALL PLISRTx, you can download and run them on your workstation. Several of the parameters allowed on S/390 are ignored, and alter run-time behavior to some extent. The following table indicates which arguments accepted by OS PL/I are ignored by the workstation compiler.

Table 34. workstation PLISRTx

Built-in subroutine	Arguments
PLISRTA Sort input: data set Sort output: data set	(sort statement,record statement,storage,return code [,data set prefix,message level, sort technique])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(sort statement,record statement,storage,return code,input routine [,data set prefix,message level,sort technique])
PLISRTC Sort input: data set Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,output routine [,data set prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,input routine,output routine[,data set prefix,message level,sort technique])

Argument definitions:

Sort statement
Character string expression describing sorting fields and format. See “Specifying the sorting field” on page 461.

Record statement
Character string expression describing the length and record format of data. See “Specifying the records to be sorted” on page 462.

Storage
Ignored by workstation PL/I.

Return code
Fixed binary variable of precision (31,0) in which sort places a return code when it has completed. The meaning of the return code is:

0=Sort successful
16=Sort failed

Input routine
(PLISRTB and PLISRTD only.) Name of the PL/I external or internal procedure used to supply the records for the Sort program at sort exit 15. For specific requirements using workstation PL/I, see “E15 — input-handling routine (sort exit E15)” on page 465.

Output routine
(PLISRTC and PLISRTD only.) Name of the PL/I external or internal procedure to which Sort passes the sorted records from sort exit 35. For specific requirements using workstation PL/I, see “E35 — output-handling routine (sort exit E35)” on page 468.

Data set prefix
Ignored by workstation PL/I, which only processes SORTIN and SORTOUT as ddnames.

Message level
Ignored by workstation PL/I.

Sort technique
Ignored by workstation PL/I.

Preparing to use sort

Before using sort, you must determine the type of sort you require, the length and format of the sorting fields in the data, and the length of your data records.

To determine which PLISRTx built-in subroutine to use, you must decide the source of your unsorted data, and the destination of your sorted data. You must choose between data sets and PL/I subroutines. Using data sets is simpler to understand and gives faster performance. Using PL/I subroutines gives you more flexibility and more function, enabling you to manipulate the data before it is sorted, and to make immediate use of the data in its sorted form. If you decide to use an input or output handling subroutine, read “Sort data handling routines” on page 465.

The sort built-in subroutines and the source and destination of data are as follows:

Built-in subroutine	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine
PLISRTD	Subroutine	Subroutine

Source data sets are defined using the SORTIN environment variable while destination data sets are defined using SORTOUT. Alternatively, you can use the PUTENV built-in function to set those functions.

Having determined the subroutine you are using, you must now determine a number of things about your data set and specify the information on the SORT statement:

- The position of the sorting fields; these can be either the complete record or any part or parts of it.
- The type of data these fields represent, for example, character or binary.
- Whether you want the sort on each field to be in ascending or descending order.

Next, you must determine two things about the records to be sorted and specify the information on the RECORD statement:

- Whether the record format is fixed or varying
- The length of the record (maximum length for varying)

You use these on the RECORD statement, which is the second argument to PLISRT*x*.

Choosing the type of sort

To make the best use of the sort program, you should understand how it works. In your PL/I program you specify a sort by using a CALL statement to the built-in subroutine PLISRTx. Each specifies a different source for the unsorted data and destination for the data when it has been sorted.

For example, a call to PLISRTA specifies that the unsorted data (the input to sort) is on a data set, and that the sorted data (the output from sort) is to be placed on another data set. The CALL PLISRT*x* statement must contain an argument list giving the sort program information about the data set to be sorted, the fields on which it is to be sorted, the name of a variable into which sort places a return code indicating the success or failure of the sort, and the name of any output or input handling procedure that can be used.

The sort interface routine builds an argument list for the sort from the information supplied by the PLISRTx argument list and depends on your choice of A, B, C, or D for x. Control is then transferred to the sort program. If you have specified an output- or input-handling routine, it is called by the sort program as many times as is necessary to handle each of the unsorted or sorted records.

The sort operation ends in one of two ways:

1. Communicating success or failure by sending a return code of 0 or 16 to the PL/I calling procedure.
2. Raising an error condition when certain errors are detected and the return code is undefined.

Figure 34 on page 460 is a simplified flowchart showing the sort operation.

Preparing to use sort

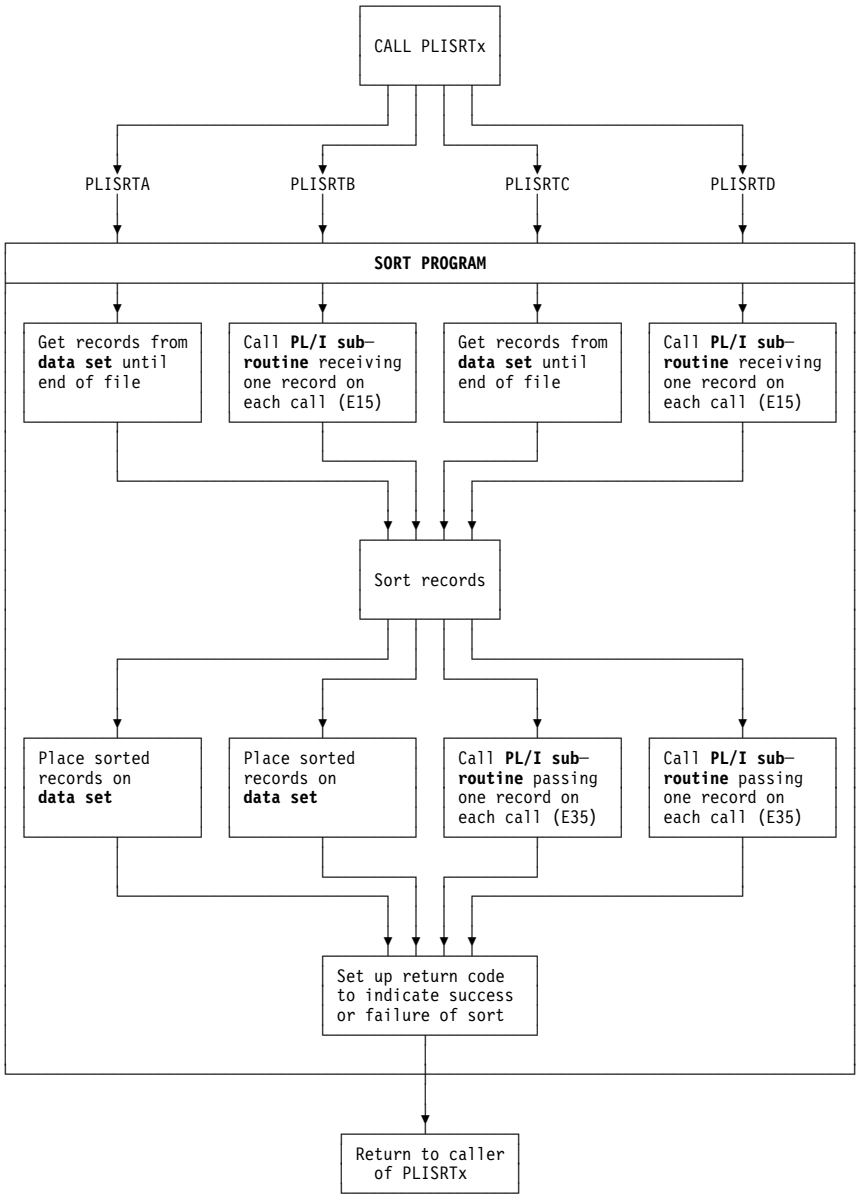


Figure 34. Flow of control for the sort program

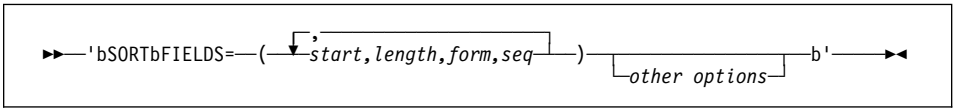
Within the sort program itself, the flow of control between the sort program and output- and input-handling routines is controlled by return codes. The sort program calls these routines at the appropriate point in its processing. (Within the sort program, these routines are known as user exits. The routine that passes input to be sorted is the E15 sort user exit. The routine that processes sorted output is the E35 sort user exit.) From the routines, the sort program expects a return code indicating either that it

should call the routine again, or that it should continue with the next stage of processing.

The remainder of this chapter gives detailed information on how to use sort from PL/I. First the required PL/I statements are described, followed by the data set requirements. The chapter finishes with a series of examples showing the use of the four built-in subroutines.

Specifying the sorting field

The SORT statement is the first argument to PLISRTx. The syntax of the SORT statement must be a character string expression that takes the form:



b One or more blanks. Blanks shown are mandatory. No other blanks are allowed.

start,length,form,seq
Sorting fields. You can specify any number of such fields, but there is a limit on the total length of the fields. If more than one field is to be sorted on, the records are sorted first according to the first field, and then those that are of equal value are sorted according to the second field, and so on. If all the sorting values are equal, the order of equal records is arbitrary. Fields can overlay each other.

start
The starting position within the record. Give the value in bytes. The first byte in a string is considered to be byte 1.

length
The length of the sorting field. Give the value in bytes. The length of sorting fields is restricted according to their data type.

form
The format of the data. This is the format assumed for the purpose of sorting. All data passed between PL/I routines and sort must be in the form of character strings. The main data types and the restrictions on their length are shown below.

Code	Data Type and Length
CH	character 1–256
ZD	zoned decimal signed 1–32
PD	packed decimal signed 1–32
FI	fixed point, signed 1–256
BI	binary, unsigned 1 bit to 256 bytes

The sum of the lengths of all fields must not exceed 256 bytes.

Preparing to use sort

seq
The sequence in which the data is sorted:

A – ascending (that is, 1,2,3,...)
D – descending (that is, ...,3,2,1).

Note that you cannot specify E, because PL/I does not provide a method of passing a user-supplied sequence.

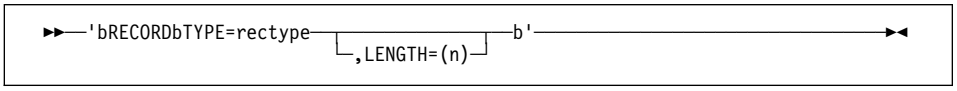
other options
The only option supported under workstation PL/I is the default, EQUALS. Source code downloaded from the mainframe, however, does not need to be altered.

Example:

```
' SORT FIELDS=(1,10,CH,A) '
```

Specifying the records to be sorted

Use the RECORD statement as the second argument to PLISRT*x*. The syntax of the RECORD statement must be a character string expression which, when evaluated, accepts the following syntax:



b One or more blanks. Blanks shown are mandatory. No other blanks are allowed.

TYPE
Specifies the type of record as follows:

F Fixed length
V Varying length

Even when you use input and output routines to handle the sorted and unsorted data, you must specify the record type as it applies to the work data sets used by sort.

If varying-length strings are passed to sort from an input routine (E15 exit), you should normally specify V as a record format. However, if you specify F, the records are padded to the maximum length with blanks.

LENGTH
Specifies the length of the record to be sorted. You can omit LENGTH if you use PLISRTA or PLISRTC, because the length is taken from the input data set. The maximum length of a record that can be sorted is 32,767 bytes. For varying-length records, you must include the 2-byte prefix.

n The length of the record to be sorted.

Note: Additional length specifications that can be used are ignored by workstation PL/I.

Example:

```
' RECORD TYPE=F,length=(80) '
```

Calling the sort program

When you have determined the sort field and record type specifications, you are in a position to write the CALL PLISRT*x* statement.

PLISRT examples

The following examples indicate commonly used forms of calls to PLISRTx.

Example 1

A call to PLISRTA sorting 80-byte records from SORTIN to SORTOUT, and a return code, RETCODE, declared as FIXED BINARY (31,0).

```
call plisrta (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                0,
                retcode);
```

Example 2

This example is the same as example 1 but the sort is to be undertaken on two fields. First, bytes 1 to 10 which are characters, and then, if these are equal, bytes 11 and 12 which contain a binary field. Both fields are to be sorted in ascending order.

```
call plisrta (' SORT FIELD =(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                0,
                retcode);
```

Example 3

A call to PLISRTB. The input is to be passed to sort by the PL/I routine PUTIN, the sort is to be carried out on characters 1 to 10 of an 80 byte fixed-length record. Other information as above.

```
call plisrtb (' SORT FIELDS=(1,10,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
                0,
                retcode,
                putin);
```

Example 4

A call to PLISRTD. The input is to be supplied by the PL/I routine PUTIN and the output is to be passed to the PL/I routine PUTOUT. The record to be sorted is 82 bytes varying (including the length prefix). It is to be sorted on bytes 1 through 5 of the data in ascending order, then if these fields are equal, on bytes 6 through 10 in descending order. If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

Sort data input and output

```
call plisrtd (' SORT FIELDS=(1,5,CH,A,6,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(82) ',
                0,
                retcode,
                putin,       /* input routine  (sort exit 15) */
                putout);     /* output routine (sort exit 35) */
```

Determining whether the sort was successful

When the sort is completed, sort sets a return code in the variable named in the fourth argument of the call to PLISRTx. It then returns control to the statement that follows the CALL PLISRTx statement. The value returned indicates the success or failure of the sort as follows:

```
0   Sort successful
16  Sort failed
```

You must declare this variable as FIXED BINARY (31,0). It is standard practice to test the value of the return code after the CALL PLISRTx statement and take appropriate action according to the success or failure of the operation.

For example (assuming the return code was called RETCODE):

```
if retcode¬=0 then do;
  put data(retcode);
  signal error;
end;
```

The error condition is raised if errors are detected. When sort detects a fatal error and the corresponding error code is greater than 16, the error condition is raised.

If the job step that follows the sort depends on the success or failure of the sort, you should set the value returned in the sort program as the return code from the PL/I program. This return code is then available for the following job step. The PL/I return code is set by a call to PLIRETC. The following example shows how you can call PLIRETC with the value returned from sort:

```
call pliretc(retcode);
```

You should not confuse this call to PLIRETC with the calls made in the input (E15) and output (E35) routines, where a return code is used for passing control information to sort.

Sort data input and output

The source of the data to be sorted is provided either directly from a data set or indirectly by a routine (sort exit E15) written by the user. Similarly, the destination of the sorted output is either a data set or a routine (sort exit E35) provided by the user.

PLISRTA is the simplest of all of the interfaces because it sorts from data set to data set. An example of a PLISRTA program is in Figure 38 on page 470. Other interfaces require either the input-handling routine or the output-handling routine, or both.

To sort varying-length records, you first need to convert your datasets to TYPE(VARLS) format, and then use this TYPE(VARLS) file as input to the sort program. TYPE(VARLS) records have a 2-byte length field at the beginning, so the record size is actually two less than the length of the record. This means the record size you specify should be two less than the maximum record length for the file.

You can convert your dataset to a TYPE(VARLS) file by writing a PL/I program that reads from the existing data file and writes to an output file declared as TYPE(VARLS).

Sort data handling routines

The input-handling and output-handling routines are called by sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRT*x*, they behave in the same way as ordinary internal procedures with respect to the scope of names. The input and output procedure names themselves must be known in the procedure that makes the call to PLISRT*x*.

The routines are called individually for each record required by sort or passed from sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures do not retain their values between calls. Consequently, items such as counters, which need to be retained from one call to the next, should either be declared as STATIC or be declared in the containing block.

E15 — input-handling routine (sort exit E15)

Input routines are normally used to process data in some way before it is sorted, such as printing it, (see Figure 39 on page 471 and Figure 41 on page 474), or generating or manipulating the sorting fields to achieve the correct results.

The input-handling routine is used by SORT when a call is made to either PLISRTB or PLISRTD. When SORT requires a record, it calls the input routine which should return a record in character string format, and a return code of 12, which means the record passed is to be included in the sort. SORT continues to call the routine until a return code of 8 is passed. This means that all records have *already* been passed, and SORT is not to call the routine again. If a record is returned when the return code is 8, it is ignored by SORT.

Note: You must compile the program that calls PLISRTB or PLISRTD with the same options (ASCII or EBCDIC; NATIVE or NONNATIVE; HEXADEC or IEEE) that you used to compile the E15 handling routine.

The data returned by the E15 routine must be a fixed or varying character string. If it is varying, you should normally specify V as the record format in the RECORD statement which is the second argument in the call to PLISRTx. However, you can specify F, in which case the string is padded to its maximum length with blanks.

The record is returned with a RETURN statement, and you must specify the RETURNS attribute in the PROCEDURE statement. The return code is set in a call to PLIRETC.

Sort data handling routines

Examples of an input routine are given in Figure 39 on page 471 and Figure 41 on page 474.

In addition to the return codes of 12 (include current record in sort) and 8 (all records sent), SORT allows the use of a return code of 16. This ends the sort and sets a return code from SORT to your PL/I program of 16–sort failed.

It should be noted that a call to PLIRETC sets a return code that is passed by your PL/I program, and is available to any job steps that follow it. When an output handling routine has been used, it is a good practice to reset the return code with a call to PLIRETC after the call to PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 40 on page 473.

```
E15: proc returns (char(80));
                        /* Returns attribute must be used specifying
                           length of data to be sorted, maximum length
                           if varying strings are passed to sort.      */

   dcl string char(80);  /* A character string variable is normally
                           required to return the data to sort         */

   if Last_Record_Sent then do;
                        /* A test must be made to see if all the
                           records have been sent, if they have, a
                           return code of 8 is set up and control
                           returned to sort                            */

        call pliretc(8); /* Set return code of 8, meaning last record
                           already sent.                               */
   end;

   else do;
                        /* If another record is to be sent to sort,
                           do the necessary processing, set a return
                           code of 12 by calling PLIRETC, and return
                           the data as a character string to sort      */

       /*    The code to do your processing goes here    */

       call pliretc (12);/* Set return code of 12, meaning this
                           record is to be included in the sort        */
       return (string);  /* Return data with RETURN statement           */
   end;
  end;                  /* End of the input procedure                  */
```

Figure 35. Skeletal code for an input procedure

In addition, to code the input user exit routine, the explicit attributes of the E15 must be specified in the program unit that calls PLISRTx if E15 is not nested in that program unit.

```
  plisort: proc options(main);

     dcl e15 entry returns(char(2000) varying);

     /* Code to do your processing goes here  */

     call plisrtb(' SORT FIELDS=(5,10,CH,A) '
                  ' RECORD TYPE=V,LENGTH=(2000) ',
                    0,
                    retcode,
                    e15);

     /* Code to do your processing goes here  */

  end plisort;

*PROCESS

E15: proc returns (char(2000) varying);
                          /* Returns option must be used specifying
                             length of data to be sorted, maximum length
                             if varying strings are passed to sort.      */

   dcl string char(2000) varying;
                          /* A character string variable is normally
                             required to return the data to sort         */

   if Last_Record_Sent then do;
                          /* A test must be made to see if all the
                             records have been sent, if they have, a
                             return code of 8 is set up and control
                             returned to sort                            */

        call pliretc(8); /* Set return code of 8, meaning last record
                             already sent.                               */
   end;

   else do;
                          /* If another record is to be sent to sort,
                             do the necessary processing, set a return
                             code of 12 by calling PLIRETC, and return
                             the data as a character string to sort      */

     /* Code to do your processing goes here  */

       call pliretc (12);/* Set return code of 12, meaning this
                             record is to be included in the sort        */
      return (string);  /* Return data with RETURN statement           */
   end;
  end;                    /* End of the input procedure                  */
```

Figure 36. When E15 is external to the procedure calling PLISRTx

Sort data handling routines

E35 — output-handling routine (sort exit E35)

You must compile the program that calls PLISRTC or PLISRTD with the same options (ASCII or EBCDIC; NATIVE or NONNATIVE) that you used to compile the E35 handling routine.

Output-handling routines are normally used for any processing that is necessary after the sort. This could be to print the sorted data, as shown in Figure 40 on page 473 and Figure 41 on page 474, or to use the sorted data to generate further information. The output handling routine is used by sort when a call is made to PLISRTC or PLISRTD.

When the records have been sorted, sort passes them (one at a time) to the output handling routine. The output routine then processes them as required. When all the records have been passed, sort sets up its return code and returns to the statement after the CALL PLISRT*x* statement. There is no indication from sort to the output handling routine that the last record has been reached. Any end-of-data handling must therefore be done in the procedure that calls PLISRT*x*.

The record is passed from sort to the output routine as a character string, and you must declare a character string parameter in the output-handling subroutine to receive the data. The output-handling subroutine must also pass a return code of 4 to sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

The sort can be stopped by passing a return code of 16 to sort. This results in sort returning to the calling program with a return code of 16–sort failed.

The record passed to the routine by sort is a character string parameter. If you specified the record type as F in the second argument in the call to PLISRT*x*, you should declare the parameter with the length of the record. If you specified the record type as V, you should declare the parameter as adjustable, for example:

```
dcl string char(*);
```

Skeletal code for a typical output-handling routine is shown in Figure 37 on page 469.

You should note that a call to PLIRETC sets a return code that is passed by your PL/I program, and is available to any job steps that follow it. When you have used an output handling routine, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples at the end of this chapter.

```
E35: proc(String);
                        /* The procedure must have a character string
                           parameter to receive the record from sort   */

  dcl String char(80);  /* Declaration of parameter                    */

 /* Your code goes here */

  call pliretc(4);      /* Pass return code to sort indicating that
                           the next sorted record is to be passed to
                           this procedure.                             */
   end E35;             /* End of procedure returns control to sort    */
```

Figure 37. Skeletal code for an output-handling procedure

Sort data handling routines

Calling PLISRTA

```
/*********************************************************************/
/*                                                                   */
/*  DESCRIPTION                                                      */
/*    Sorting from an input data set to an output data set           */
/*                                                                   */
/*  Use the following statements:                                    */
/*    set dd:sortin=ex106.dat,type(crlf),lrecl(80)                   */
/*    set dd:sortout=ex106.out,type(crlf),lrecl(80)                  */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

  ex106: proc options(main);
      dcl Return_code fixed bin(31,0);

      call plisrta (' SORT FIELDS=(7,74,CH,A) ',
                    ' RECORD TYPE=F,LENGTH=(80) ',
                      0,
                      Return_code);
      select (Return_code);
        when(0) put skip edit
            ('Sort complete return_code 0') (a);
        when(16) put skip edit
            ('Sort failed, return_code 16') (a);
        other    put skip edit (
             'Invalid sort return_code = ', Return_code) (a,f(2));
      end /* Select */;
      /* Set pl/i return code to reflect success of sort */
      call pliretc(Return_code);
  end ex106;
```

Figure 38. PLISRTA—Sorting from input data set to output data set

Content of EX106.DAT to be used with Figure 38

```
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
```

Calling PLISRTB

```
/*******************************************************************/
/*                                                                 */
/*  DESCRIPTION                                                    */
/*    Sorting from an input-handling routine to an output data set */
/*                                                                 */
/*  Use the following statements:                                  */
/*                                                                 */
/*    set dd:sysin=ex107.dat,type(crlf),lrecl(80)                  */
/*    set dd:sortout=ex107.out,type(crlf),lrecl(80)                */
/*                                                                 */
/*******************************************************************/

 ex107:  proc options(main);

     dcl Return_code fixed bin(31,0);

     call plisrtb (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  0,
                  Return_code,
                  e15x);
     select(Return_code);
       when(0)  put skip edit
            ('Sort complete return_code 0') (a);
       when(16) put skip edit
            ('Sort failed, return_code 16') (a);
       other   put skip edit
            ('Invalid return_code = ',Return_code)(a,f(2));
     end /* Select */;
         /* Set pl/i return code to reflect success of sort    */
     call pliretc(Return_code);

 e15x:   /* Input-handling routine gets records from the input
            stream and puts them before they are sorted        */
     proc returns (char(80));
          dcl sysin file stream input,
              Infield char(80);

          on endfile(sysin) begin;
             put skip(3) edit ('End of sort program input')(a);
             call pliretc(8);  /* Signal that last record has
                                  already been sent to sort    */
             goto ende15;
             end;

           get file (sysin) edit (infield) (l);
           put skip edit (infield)(a(80)); /*  Print input     */
           call pliretc(12);  /* Request sort to include current
                                 record and return for more    */
          return(Infield);
    ende15:
          end e15x;
 end ex107;
```

Figure 39. PLISRTB—Sorting from input-handling routine to output data set

Sort data handling routines

Content of EX107.DAT to be used with Figure 39 on page 471

```
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
```

Calling PLISRTC

```
/*******************************************************************/
/*                                                                 */
/*  DESCRIPTION                                                    */
/*    Sorting from an input data set to an output-handling routine */
/*                                                                 */
/*  Use the following statement:                                   */
/*                                                                 */
/*    set dd:sortin=ex108.dat,type(crlf),lrecl(80)                 */
/*                                                                 */
/*******************************************************************/

 ex108:  proc options(main);

     dcl Return_code fixed bin(31,0);

     call plisrtc (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   0,
                   Return_code,
                   e35x);
        select(Return_code);
          when(0)  put skip edit
              ('Sort complete return_code 0') (a);
          when(16) put skip edit
              ('Sort failed, return_code 16') (a);
          other    put skip edit
              ('Invalid return_code = ', Return_code) (a,f(2));
        end /* Select */;
      /* Set pl/i return code to reflect success of sort       */
     call pliretc (return_code);

 e35x:   /* Output-handling routine prints sorted records      */
       proc (Inrec);
          dcl inrec char(*);
          put skip edit (inrec) (a);
          call pliretc(4); /* Request next record from sort    */
       end e35x;
 end ex108;
```

Figure 40. PLISRTC—Sorting from input data set to output-handling routine

Content of EX108.DAT to be used with Figure 40

```
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
```

Sort data handling routines

Calling PLISRTD, example 1

```
/*****************************************************************/
/*                                                               */
/*  DESCRIPTION                                                  */
/*    Sorting an input-handling to output-handling routine       */
/*                                                               */
/*  Use the following statement:                                 */
/*                                                               */
/*    set dd:sysin=ex109.dat,type(crlf),lrecl(80)                */
/*                                                               */
/*****************************************************************/

 ex109:  proc options(main);
     dcl Return_code fixed bin(31,0);
     call plisrtd (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   0,
                   Return_code,
                   e15x,
                   e35x);

     select(Return_code);
       when(0)  put skip edit
           ('Sort complete return_code 0') (a);
       when(16) put skip edit
           ('Sort failed, return_code 16') (a);
       other    put skip edit
           ('Invalid return_code = ', Return_code) (a,f(2));
       end /* select */;

      /* Set pl/i return code to reflect success of sort        */
     call pliretc(Return_code);

 e15x:   /* Input-handling routine prints input before sorting  */
       proc returns(char(80));
            dcl infield char(80);

            on endfile(sysin) begin;
               put skip(3) edit ('end of sort program input.  ',
                   'sorted output should follow')(a);
               call pliretc(8);  /* Signal end of input to sort */
               goto ende15;
            end;

            get file (sysin) edit (infield) (l);
            put skip edit (infield)(a);
            call pliretc(12);  /* Input to sort continues       */
           return(Infield);
  ende15:
            end e15x;

  e35x:   /* Output-handling routine prints the sorted records  */
       proc (Inrec);
            dcl inrec char(80);
            put skip edit (inrec) (a);
     next:  call pliretc(4); /* Request next record from sort   */
            end e35x;
 end ex109;
```

Figure 41. PLISRTD—Sorting input-handling routine to output-handling routine

Contents of EX109.DAT and EX110.DAT used with Figure 41 on page 474 and Figure 42 on page 476

```
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
```

Calling PLISRTD, example 2

```
ex110: proc options(main);

  /*****************************************************************/
  /*                                                               */
  /* PLISRTD: sorting from an input-handling rtn to an             */
  /*          output-handling routine. Records are varying-length. */
  /*                                                               */
  /*****************************************************************/

  dcl rc fixed bin(31,0);

  call plisrtd(' SORT FIELDS=(7,4,CH,A) ',
               ' RECORD TYPE=V,LENGTH=(80) ',
                 256000,
                 rc,
                 e15x,
                 e35x );

  select( rc );
    when(0) put skip edit
      ('Sort complete return code = 0') (a);
    when(16) put skip edit
      ('Sort failed return code = 16') (a);
    other put skip edit
      ('Invalid return code = ', rc) (a,f(2));
  end;

  call pliretc(rc);


  e15x: proc returns( char(80) varying );

     dcl infield char(80) var;

     on endfile(sysin) begin;
       put skip(3) edit('End of sort program input.  ',
                        'Sortout output should follow') (a);
       call pliretc(8);
       goto ende15;
     end;

     get file(sysin) edit(infield) (l);
     put skip edit( infield ) (a);
     call pliretc(12);

     return(infield);
   ende15:
  end e15x;

  e35x: proc ( inrec );

     dcl inrec char(*);

     put skip edit(inrec) (a);
     call pliretc(4);

  end e35x;
end ex110;
```

Figure 42. PLISRTD—Sorting input-handling routine to output-handling routine

Chapter 28. 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 be instead be accomplished by PL/I program logic.

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 and OS/390 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:

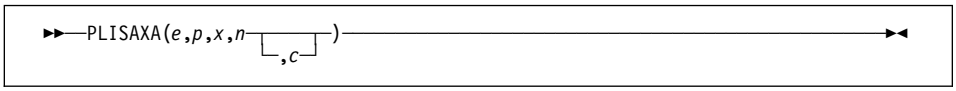
- It provides high-performance, but non-standard interfaces.
- It supports XML files encoded in either Unicode UTF-16 or any of several single-byte code pages listed below.
- The parser is non-validating, but does partially check well-formedness. See section 2.5.10,

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.

The PLISAXA built-in subroutine

The PLISAXA built-in subroutine allows you to invoke the XML parser for an XML document residing in a buffer in your program.



e An event structure

p A pointer value or "token" that the parser will pass back to the event functions

x The address of the buffer containing the input XML

n The number of bytes of data in that buffer

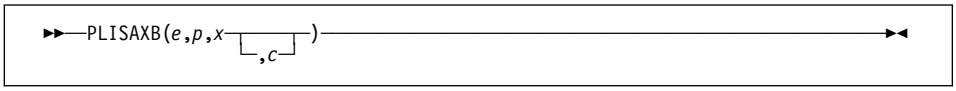
c The purported codepage of that XML

Note that if the XML is contained in a CHARACTER VARYING or a WIDECHAR VARYING string, then the ADDRDATA built-in function should be used to obtain the address of the first data byte.

Also note that if the XML is contained in a WIDECHAR string, the value for the number of bytes is twice the value returned by the LENGTH built-in function.

The PLISAXB built-in subroutine

The PLISAXB built-in subroutine allows you to invoke the XML parser for an XML document residing in a file.



e An event structure

p A pointer value or "token" that the parser will pass back to the event functions

x A character string expression specifying the input file

c The purported codepage of that XML

The SAX event structure

The event structure is a structure consisting of 24 LIMITED ENTRY variables which point to functions that the parser will invoke for various "events".

The descriptions below of each event refer to the example of an XML document in Figure 43. 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&apos;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 43. 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 );

  define alias event_end_of_document
      limited entry( pointer )
      returns( byvalue fixed bin(31) )
      options( byvalue );

  define alias event_predefined_ref
      limited entry( pointer, char(1) )
      returns( byvalue fixed bin(31) )
      options( byvalue nodescriptor );

  define alias event_character_ref
      limited entry( pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue );

  define alias event_pi
      limited entry( pointer, pointer, fixed bin(31),
                     pointer, fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue );

  define alias event_exception
      limited entry( pointer, pointer, fixed bin(31),
                     fixed bin(31) )
      returns( byvalue fixed bin(31) )
      options( byvalue );
```

Figure 44. 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 45. PLISAXA coding example - event structure

```
   dcl token       char(8);

   dcl xmlDocument char(4000) var;

   xmlDocument =
      '<?xml version="1.0" standalone="yes"?>'
   || '<!--This document is just an example-->'
   || '<sandwich>'
   || '<bread type="baker&apos;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 46. PLISAXA coding example - main routine

```
dcl chars char(32000) based;

start_of_document:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' length=' || tokenlength );

  return(0);
end;

version_information:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

encoding_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 47 (Part 1 of 8). PLISAXA coding example - event routines

```
standalone_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

document_type_declaration:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

end_of_document:
  proc( userToken )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;

  put skip list( lowercase( procname() ) );

  return(0);
end;
```

Figure 47 (Part 2 of 8). PLISAXA coding example - event routines

```
start_of_element:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

attribute_name:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

attribute_characters:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 47 (Part 3 of 8). PLISAXA coding example - event routines

```
attribute_predefined_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue nodescriptor );

  dcl userToken      pointer;
  dcl reference   char(1);

  put skip list( lowercase( procname() )
   || ' ' || hex(reference ) );

  return(0);
end;

attribute_character_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl reference      fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || hex(reference ) );

  return(0);
end;

end_of_element:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 47 (Part 4 of 8). PLISAXA coding example - event routines

```
start_of_CDATA:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

end_of_CDATA:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

content_characters:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 47 (Part 5 of 8). PLISAXA coding example - event routines

```
content_predefined_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue nodescriptor );

  dcl userToken      pointer;
  dcl reference   char(1);

  put skip list( lowercase( procname() )
   || ' ' || hex(reference ) );

  return(0);
end;

content_character_reference:
  proc( userToken, reference )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl reference      fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || hex(reference ) );

  return(0);
end;

processing_instruction:
  proc( userToken, piTarget, piTargetLength,
                           piData, piDataLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl piTarget       pointer;
  dcl piTargetLength fixed bin(31);
  dcl piData         pointer;
  dcl piDataLength   fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(piTarget->chars,1,piTargetLength ) || '>' );

  return(0);
end;
```

Figure 47 (Part 6 of 8). PLISAXA coding example - event routines

```
comment:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

unknown_attribute_reference:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

unknown_content_reference:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;
```

Figure 47 (Part 7 of 8). PLISAXA coding example - event routines

```
start_of_prefix_mapping:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

end_of_prefix_mapping:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
   || ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

  return(0);
end;

exception:
  proc( userToken, xmlToken, currentOffset, errorID )
  returns( byvalue fixed bin(31) )
  options( byvalue );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl currentOffset  fixed bin(31);
  dcl errorID        fixed bin(31);

  put skip list( lowercase( procname() )
   || ' errorid =' || errorid );

  return(0);
end;
```

Figure 47 (Part 8 of 8). PLISAXA coding example - event routines

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 48. 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 35 (Page 1 of 5). 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.

Table 35 (Page 2 of 5). Continuable Exceptions

Number	Description	Parser Action on Continuation
2	The parser found an invalid start of a processing instruction, element, comment or document type declaration outside element content.	The parser generates a content_characters event with the XML text containing the 2- or 3-character invalid initial character sequence. Parsing continues at the character after the invalid sequence.
3	The parser found a duplicate attribute name.	The parser generates an attribute_name event with the XML text containing the duplicate attribute name.
4	The parser found the markup character "<" in an attribute value.	Prior to generating the exception event, the parser generates an attribute_characters event for any part of the attribute value prior to the "<" character. After the exception event, the parser generates an attribute_characters event with XML text containing "<". Parsing then continues at the character after the "<".
5	The start and end tag names of an element did not match.	The parser generates an end_of_element event with XML text containing the mismatched end name.
6	The parser found an invalid character in element content.	The parser includes the invalid character in XML text for the subsequent content_characters event.
7	The parser found an invalid start of an element, comment, processing instruction or CDATA section in element content.	Prior to generating the exception event, the parser generates a content_characters event for any part of the content prior to the "<" markup character. After the exception event, the parser generates a content_characters event with XML text containing 2 characters: the "<" followed by the invalid character. Parsing continues at the character after the invalid character.
8	The parser found in element content the CDATA closing character sequence “]]” without the matching opening character sequence “<![CDATA[”.	Prior to generating the exception event, the parser generates a content_characters event for any part of the content prior to the “]]” character sequence. After the exception event, the parser generates a content_characters event with XML text containing the 3-character sequence “]]”. Parsing continues at the character after this sequence.
9	The parser found an invalid character in a comment.	The parser includes the invalid character in XML text for the subsequent comment event.
10	The parser found in a comment the character sequence "--" not followed by ">".	The parser assumes that the "--" character sequence terminates the comment, and generates a comment event. Parsing continues at the character after the "--" sequence.
11	The parser found an invalid character in a processing instruction data segment.	The parser includes the invalid character in XML text for the subsequent processing_instruction event.

Table 35 (Page 3 of 5). Continuable Exceptions

Number	Description	Parser Action on Continuation
12	A processing instruction target name was "xml" in lower-case, upper-case or mixed-case.	The parser generates a processing_instruction event with XML text containing "xml" in the original case.
13	The parser found an invalid digit in a hexadecimal character reference (of the form &#xdddd;).	The parser generates an attribute_characters or content_characters event with XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.
14	The parser found an invalid digit in a decimal character reference (of the form &#dddd;).	The parser generates an attribute_characters or content_characters event with XML text containing the invalid digit. Parsing of the reference continues at the character after this invalid digit.
15	The encoding declaration value in the XML declaration did not begin with lower- or upper-case A through Z	The parser generates the encoding event with XML text containing the encoding declaration value as it was specified.
16	A character reference did not refer to a legal XML character.	The parser generates an attribute_character_reference or content_character_reference event with XML-NTEXT containing the single Unicode character specified by the character reference.
17	The parser found an invalid character in an entity reference name.	The parser includes the invalid character in the XML text for the subsequent unknown_attribute_reference or unknown_content_reference event.
18	The parser found an invalid character in an attribute value.	The parser includes the invalid character in XML text for the subsequent attribute_characters event.
50	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
51	The document was encoded in EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
52	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an ASCII encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.

Table 35 (Page 4 of 5). Continuable Exceptions

Number	Description	Parser Action on Continuation
53	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
54	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
55	The document was encoded in EBCDIC, and the CODEPAGE compiler option specified a supported EBCDIC code page, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
56	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
57	The document was encoded in ASCII, and the document encoding declaration specified a supported ASCII encoding, but the parser does not support the code page specified by the CODEPAGE compiler option.	The parser uses the encoding specified by the document encoding declaration.
58	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported EBCDIC encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
59	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CODEPAGE compiler option.
60	The document was encoded in ASCII, and the CODEPAGE compiler option specified a supported ASCII code page, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CODEPAGE compiler option.
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.

Table 35 (Page 5 of 5). Continuable Exceptions

Number	Description	Parser Action on Continuation
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 36 (Page 1 of 4). Terminating Exceptions

Number	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.

Table 36 (Page 2 of 4). Terminating Exceptions

Number	Description
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, '_' or ':'.
125	The first character of the first attribute name of an element was not a letter, '_' or ':'.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than '=' following an attribute name.
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, '_' or ':'.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a '>' following the '/'.
133	The first character of an element end tag name was not a letter, '_' or ':'.
134	An element end tag name was not terminated by a '>'.
135	The first character of an element name was not a letter, '_' or ':'.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.
138	The first character of a processing instruction target name was not a letter, '_' or ':'.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence '?>'.
141	The parser found an invalid character following '&' in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by a '='.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.

Table 36 (Page 3 of 4). Terminating Exceptions

Number	Description
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.
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.

Table 36 (Page 4 of 4). Terminating Exceptions

Number	Description
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.
500 to 99,999	Internal error. Please report the error to your service representative.

Chapter 29. Remote IMS DL/I support

Remote DL/I provides the support to develop and test mainframe PL/I programs that use DL/I calls in a subset of environments. Specifically, Remote DL/I provides access to IMS full function databases and GSAM databases from programs using DL/I calls.

Configure your system

Before you use Remote DL/I, you should make sure that your system is configured properly. The product readme file contains instructions on finding more detailed configuration information.

If you use PLITDLI, you can develop, compile, and test PL/I programs that run in an IMS batch environment and use PLITDLI calls.

Remote DL/I does not provide access to IMS message queues or IMS fast path databases. Remote DL/I runs using only S/390 data types as input and output. It does not provide any data conversion functions.

How Remote DL/I works

Remote DL/I uses APPC and an IMS batch environment to provide the remote DL/I call support. When Remote DL/I is first initialized on the workstation, it asks you for the TSO userid and password to be used when the remote job is started on MVS.

APPC is used by Remote DL/I to start a job on MVS and bring up an IMS batch environment. Once the IMS batch environment is available, remote DL/I calls can be processed.

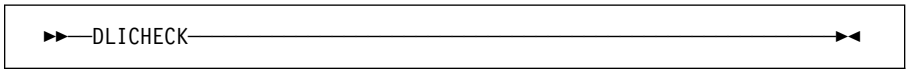
DL/I calls on the workstation are sent to the IMS batch environment and executed. The results of the DL/I call are then sent back to the program running on the workstation.

Remote DL/I utilities

Here is a list of utility commands provided by Remote DL/I.

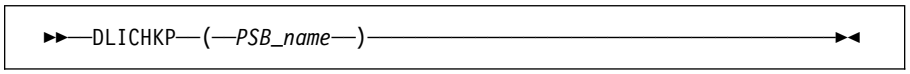
DLICHECK Command

Verifies that your Remote DL/I connection is working.



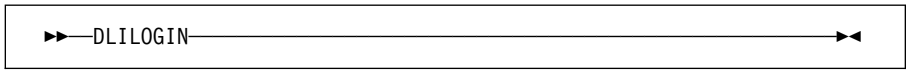
DLICHKP Command

Verifies that your Remote DL/I connection is working and that a specific PSB can be successfully scheduled and terminated.



IMS batch support

DLILOGIN Command
Updates the userid and password used by Remote DL/I.

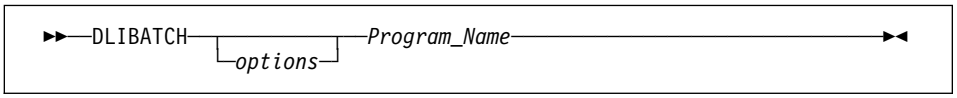


IMS batch support

This section describes the support that allows a programmer to develop and test S/390 IMS Batch PL/I programs.

Introducing the DLIBATCH command

Use the DLIBATCH command to invoke IMS batch PL/I programs.



options

- */E:Entry_Name*—Name of the entry point in *Program_Name* where program execution begins. If this option is not specified, it defaults to *Program_Name*.
- */P:PSB_Name*—Name of the PSB to schedule. If this option is not specified, it defaults to *Program_Name*.
- */D:Debugger_Name*—Name of the debugger to be given control before starting the program. If this option is not specified, program execution begins at the *Entry_Name* or its default. When using PL/I, PDBUG is the only valid value for *Debugger_Name*.

Program_name
Must be the name of a DLL.

The DLIBATCH command supports PSBs with up to 100 PCBs.

Preparing to use the DLIBATCH command

In order to use the DLIBATCH command, consider the following:

- The programs that are going to be run must be compiled and link edited with certain options. For example, since Remote DL/I runs using S/390 data types as input and output, you must use specific compiler options that enable S/390 data type support (see “Compiling VisualAge PL/I programs on Windows” on page 509).
- The Remote DL/I server environment file or the JCL associated with the APPC TP profile that is used to bring up the remote IMS batch environment might need to be modified. For example, if a new PSB was created, the dataset that contains the new PSB must be included in the IMS DD statement. For information on the server environment file, see “Remote DL/I server environment file” on page 510.

- The RMTDLI_PARTNER_LU and the RMTDLI_PARTNER_TP environment variables must be properly set.
- When programs access files using native language (such as OPEN, READ, WRITE, etc), you must set up access to those files. You can set up the files as local or remote files (remote file access is provided by SMARTdata Utilities).

Compiling VisualAge PL/I programs on Windows

WIN Use the following PL/I compile-time options for Remote DL/I:

- XINFO(DEF)
- DEFAULT(EBCDIC NONNATIVE HEXADEC DESCRIPTOR LINKAGE(OPTLINK) SYSTEM(IMS) IMS(BYVALUE))

Compiling VisualAge PL/I programs on OS/2

OS/2 Use the following PL/I compile-time options for Remote DL/I:

DEFAULT(EBCDIC NONNATIVE HEXADEC DESCRIPTOR LINKAGE(SYSTEM) SYSTEM(IMS) IMS(BYVALUE))

Linking VisualAge PL/I programs

In order for an IMS PL/I program to be called by the DLIBATCH command, the program must be in a DLL. See Chapter 21, “Building dynamic link libraries” on page 386 for instructions on creating a DLL.

Using the DLIBATCH command

Once you have completed the preparatory steps, you can use the DLIBATCH command while considering the following items.

MVS userid and password

If Remote DL/I is not already running, the DLIBATCH command will cause Remote DL/I to become active. The first time Remote DL/I is used on a workstation, you should see a prompt requesting the MVS userid and password needed to start the server job. The password information is saved until the workstation is shutdown.

The userid and password information can be updated using the DLILOGIN command.

Invoking the target program

Once DLIBATCH has successfully started an IMS batch environment on MVS with the specified PSB name, it invokes the target program on the workstation with the PCBs obtained from the IMS batch environment.

Supported function codes

All of the function codes that are supported in an IMS batch environment are supported when using DLIBATCH except GSCD.

Syncpoint coordination

IMS/ESA provides syncpoint control with DB2 in an IMS batch environment. When using the DLIBATCH command, Remote DL/I does not provide any capability on the workstation to provide syncpoint coordination with DB2.

Server environment file

Diagnostics

Error messages produced by Remote DL/I have a prefix of IWZ. If there are any errors detected by DLIBATCH, messages are written to the system logical output device (the terminal) and the command ends.

In some cases, error messages indicate that there was some sort of problem on the server (for example, a JCL error). On the server, you need to look at the MVS message log and the Remote DL/I message log. The MVS messages are written to the message data set defined in the TP profile. The Remote DL/I messages are written to the DD IWZRDOUT specified in the JCL in the TP profile.

Using a debugger

DLIBATCH can be used with the VisualAge PL/I debugger PDBUG. To use the debugger, you must specify the */D:* option of the DLIBATCH command.

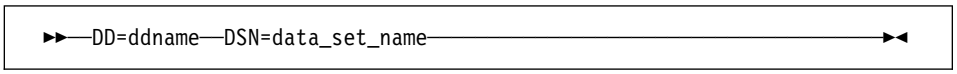
Remote DL/I server environment file

This section describes how you can alter the Remote DL/I server environment using a file on the workstation.

Within a server environment file, you can specify DD names and the associated dataset names that you want the Remote DL/I server to allocate before bringing up the IMS batch environment.

Remote DL/I gets the name of the server environment file from the environment variable RMTDLI_SERVER_ENV.

Use the following syntax to specify a DD name and associated dataset name:



Usage notes:

1. Both the DD name and the dataset name must be on the same line.
2. The keywords, the *ddname*, and the *data_set_name* must be capitalized.
3. DISP=SHR is used when the dataset is allocated.
4. Concatenation is supported only when the DD name is IMS. If more than one library is required for the IMS DD, then there needs to be one line per dataset.
5. There is no support to provide anything but a dataset name (for example, there is no support for DD DUMMY or SYSOUT).
6. If the DD name specified in the server environment file is also specified in the JCL for the Remote DL/I TP profile, the dynamic allocation will fail.

Figure 49 on page 511 is an example of a server environment file.

```
DD=RDLIDSN  DSN=IMS.DATABASE.RDLIDSN
DD=RDLIDSNO DSN=IMS.DATABASE.RDLIDSNO
DD=IMS      DSN=MYTEST.PSBLIB
DD=IMS      DSN=IMS.PSBLIB
DD=IMS      DSN=IMS.DBDLIB
```

Figure 49. Example Remote DL/I server environment file

Checkpoint and rollback support

Remote DL/I can be set up to support the ability to checkpoint and rollback database updates by enabling the batch environment as follows:

- The IEFRDER DD card must specify a system log that is on direct access storage.
- Specify dynamic backout using the IMS batch BKO execution parameter.

If the IMS batch environment is not set up as stated correctly, rollback calls get an 'AL' status code from IMS. There is no requirement that the PSB be generated with CMPAT=YES in order to get checkpoint and rollback support.

Figure 50 gives an example of the JCL used in a TP profile to enable Remote DL/I for checkpoint and rollback.

```
//RMTDLI   JOB
//*********************************************************************
//* Remote DL/I Server
//*
//*********************************************************************
//RMTDLI    EXEC PGM=IWZRDM01,REGION=8M,
//   PARM='DLI,PGMNAME,PSBNAME,,0000,,,,,,,,N,N,,Y'
//*                                                  |
//* BKO parameter--------------------------------+
//*
//STEPLIB   DD DISP=SHR,DSN=REMOTE.DLI.LOADLIB
//          DD DISP=SHR,DSN=CEEV1R50.SCEERUN
//          DD DISP=SHR,DSN=IMSVS.IMS5.RESLIB
//DFSRESLB  DD DISP=SHR,DSN=IMSVS.IMS5.RESLIB
//IMS       DD DISP=SHR,DSN=IMS.PSBLIB
//          DD DISP=SHR,DSN=IMS.DBDLIB
//IEFRDER   DD DSN=IMS.RMTDLI.LOG,DISP=SHR        << System log to DASD
//IEFRDER2  DD DUMMY
//SYSUDUMP  DD SYSOUT=*
//DFSVSAMP  DD DISP=SHR,DSN=IMS.DFSVSAMP
//* IMS databases
//RDLIDSN   DD DISP=SHR,DSN=IMS.RDLI.RDLIDSN
//RDLIDSNO  DD DISP=SHR,DSN=IMS.RDLI.RDLIDSNO
//* DD Statements required by Remote IMS
//IWZRDOUT  DD SYSOUT=*
```

Figure 50. Sample JCL for a TP profile enabled for checkpoint and rollback

Chapter 30. Using PL/I MLE in your applications

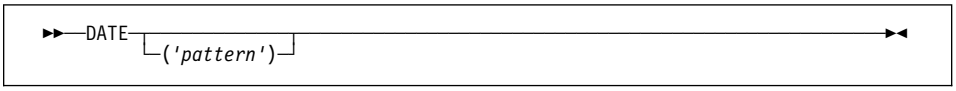
With the introduction of MLE, VisualAge PL/I allows support for a number of additional language features. The purpose of this chapter is for you to become familiar with the new attribute, compile-time options, date patterns, and built-in functions. As you follow the sequence of the chapter, you should have an idea about how to apply these to your existing applications.

Applying attributes and options

The language features introduced in these sections are found elsewhere in the Programming Guide and Language Reference, but are repeated in this chapter so you can better understand how they work together.

DATE attribute

Implicit date comparisons and conversions are made by the compiler if the two operands have the DATE attribute. The DATE attribute specifies that a variable, argument, or returned value holds a date with a specified pattern. &mlef. supports a number of date patterns as described in “Understanding date patterns” on page 515.



pattern

One of the supported date patterns. If you do not specify a pattern, YYMMDD is the default.

The DATE attribute is valid only with variables having one of the following sets of attributes:

- CHAR(n) NONVARYING
- PIC'(n)9' REAL
- FIXED DEC(n,0) REAL

The length or precision, *n*, must be a constant equal to the length of the date pattern or default pattern.

When the RESPECT compile-time option (discussed later in this chapter) has been specified, the DATE built-in function returns a value that has the attribute DATE('YYMMDD'). This allows DATE() to be assigned to a variable with the attribute DATE('YYMMDD') without an error message being generated. If DATE() is assigned to a variable not having the DATE attribute, however, an error message is generated.

Here are a few examples using the DATE attribute:

```
dcl gregorian_Date char(6) date;

dcl julian_Date    pic'(5)9' date ('YYDDD');

dcl year           fixed dec(2) date('YY');
```

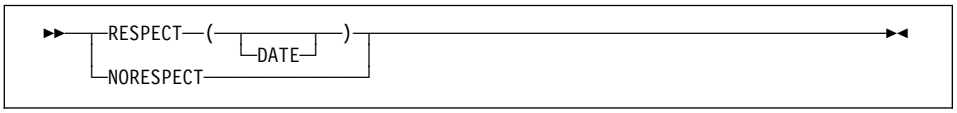
RESPECT compile-time option

The DATE attribute is useful even if you have no year 2000 problems in your applications. You can use it to manipulate differing dates as shown in these examples:

```
dcl gregorian_Date   char(8) date ('YYYYMMDD');

dcl julian_Date      pic'(7)9' date ('YYYYDDD');

if julian_Date > gregorian_Date then ...
```

RESPECT compile-time option

Use the RESPECT option to specify which attributes the compiler should recognize. Currently, DATE is the only selection possible for this compile-time option.



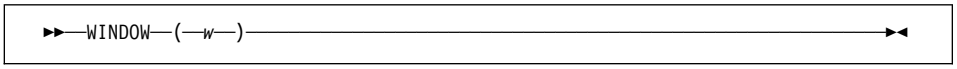
The default is RESPECT() and causes the compiler to ignore any specification of the DATE attribute. Therefore, the DATE attribute is not applied to the result of DATE built-in. NORESPECT is a synonym for RESPECT()

Specifying RESPECT(DATE), on the other hand, causes the compiler to honor any specification of the DATE attribute and to apply the DATE attribute to the result of DATE built-in.

RESPECT() is not accepted when compiling with the `PLI` command on TSO/MVS.

WINDOW compile-time option

By default, all dates with two-digit years are viewed as falling in a window starting with 1950 and ending in 2049. You can use the WINDOW option to change the value for your century window.



As previously mentioned, the default for this option is WINDOW(1950). You can specify the value for *w* as one of the following:

- An unsigned integer between 1582 and 9999 (inclusive) that represents the start of a fixed century window
- A negative integer between -1 and -99 (inclusive) that creates a "sliding" century window
- Zero, indicating the value for `w` is the current year.

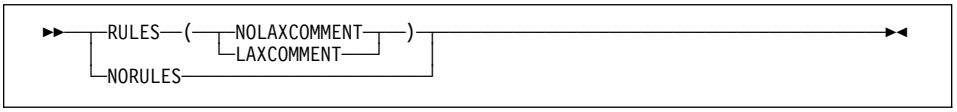
To create a fixed window, you could specify WINDOW(1900) and all two-digit years would be assumed to occur in the 20th century.

If the current year were 1998, and you wanted to create a sliding window, you could specify WINDOW(-5). The resulting century window would span the years 1993 through 2092, inclusive. When the year changes to 1999, the window would also move forward by one year.

If you set a value for the century window using the WINDOW compile-time option, that value is used for the window argument in the built-in functions which allow it, unless otherwise specified in that built-in. See “Using built-in functions with MLE” on page 517 for more details.

RULES compile-time option

In general, the RULES option allows or disallows certain language capabilities and allows you to choose semantics when alternatives are available. Currently, LAXCOMMENT is the only selection available for this option.



The default is RULES(NOLAXCOMMENT). LAXCOM and NOLAXCOM are acceptable abbreviations for the suboptions.

If you specify RULES(LAXCOMMENT), the compiler ignores the special characters /*/; therefore, whatever comes between the sets of characters is interpreted as part of the syntax instead of as a comment. If you specify RULES(NOLAXCOMMENT), the compiler treats /*/ as the start of a comment which continues until a closing */ is found.

If you happen to have workstation code that you are porting to the mainframe and uses /*/ around the DATE attribute, you need to use the RULES(LAXCOMMENT) option so that the compiler honors the attribute.

Understanding date patterns

PL/I MLE supports a series of date patterns as shown in the following table.

Table 37 (Page 1 of 2). Date patterns supported by PL/I MLE

	4-digit year	Example	2-digit year	Example
Year first	YYYY	1999	YY	99
	YYYYMM	199912	YYMM	9912
	YYYYMMDD	19991225	YYMMDD	991225
	YYYYMMM	1999DEC	YYMMM	99DEC
	YYYYMMMDD	1999DEC25	YYMMMDD	99DEC25
	YYYYMmm	1999Dec	YYMmm	99Dec
	YYYYMmmDD	1999Dec25	YYMmmDD	99Dec25
	YYYYDDD	1999359	YYDDD	99359

Table 37 (Page 2 of 2). Date patterns supported by PL/I MLE

	4-digit year	Example	2-digit year	Example
Month first	MMYYYY	121999	MMYY	1299
	MMDDYYYY	12251999	MMDDYY	122599
	MMMYYYY	DEC1999	MMMYY	DEC99
	MMMDDYYYY	DEC251999	MMMDDYY	DEC2599
	MmmYYYY	Dec1999	MmmYY	Dec99
	MmmDDYYYY	Dec251999	MmmDDYY	Dec2599
Day first	DDMMYYYY	25121999	DDMMYY	251299
	DDMMMYYYY	25DEC1999	DDMMMYY	25DEC99
	DDMmmYYYY	25Dec1999	DDMmmYY	25Dec99
	DDDYYYY	3591999	DDDYY	35999

When the day or month is omitted from one of these patterns, the compiler assumes it has a value of 1.

If the day or month are not omitted but out of range, for example 00/38/11, a message is issued if the date involves a comparison. Exceptions to the rules are cases of patterns `YYMM` and `YYMMDD` with values of all zeros that will be converted to a Julian date of 1, that, is, the smallest valid date.

Patterns and windowing

To define how a date with a two-digit year (`YY`) is interpreted, a century window is defined using the WINDOW compile-time option. As described previously, the century window defines the beginning of a 100-year span to which the two-digit year applies.

Without the help of PL/I's Millennium Language Extensions, you would have to implement something like the following logic which converts `y2` from a two-digit year to a four-digit year with a window (`w`).

```
dcl y4 pic'9999';
dcl cc pic'99';

cc = w/100;

if y2 < mod(w,100) then
  y4 = (100 * cc) + 100 + y2;
else
  y4 = (100 * cc) + y2;
```

Using this example, if you were to specify WINDOW(1900), 19 would be interpreted as the year 1919. If you were to specify WINDOW(1950), however, 19 would be interpreted as the year 2019.

Conversely, this logic calculates the two-digit year (`y2`) when converting from a four-digit year.

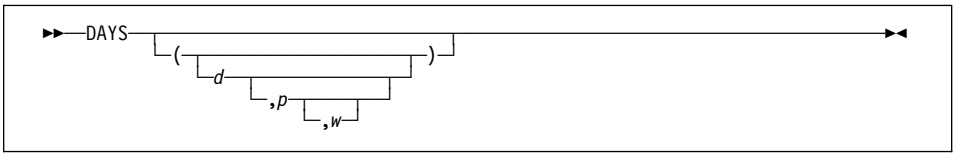
```
dcl y4 pic'9999';

if y4 < w | y4 >= w + 100 then
    signal error;

y2 = mod(y4,100);
```

Using built-in functions with MLE

The date patterns for PL/I MLE are supported by the DAYS and DAYSTODATE built-in functions. These built-ins both accept the optional argument (w) that specifies a window to be used in handling two-digit year patterns. If you specify w as part of DAYS or DAYSTODATE, the value you enter overrides the value as defined by the WINDOW compile-time option.

DAYS

DAYS returns a FIXED BINARY(31,0) value which is the number of days (in Lilian format) corresponding to the date *d*.



d String expression representing a date. If omitted, it is assumed to be the value returned by DATETIME().

The value for *d* should have character type. If not, *d* is converted to character.

p One of the supported date patterns shown in Table 37 on page 515. If omitted, the compiler assumes that *p* is the default pattern returned by the DATETIME built-in function (YYYYMMDDHHMISS999).

p should have character type. If not, it is converted to character.

w An integer expression that defines a century window to be used to handle any two-digit year formats.

- If the value is positive, such as 1950, it is treated as a year.
- If negative or zero, the value specifies an offset to be subtracted from the current, system-supplied year.
- If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

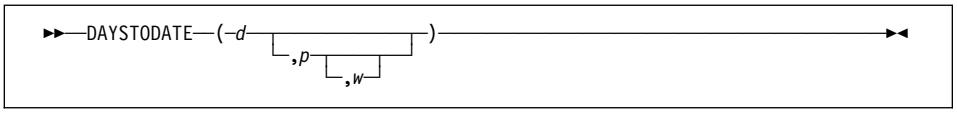
DAYSTODATE

The following example shows uses of both the DAYS and DAYSTODATE built-in functions:

```
dcl date_format char(8) static init('MMDDYYYY');
dcl todays_date char(8);
dcl sep2_1993 char(8);
dcl days_of_july4_1993 fixed bin(31);
dcl msg char(100) varying;
dcl date_due char(8);

todays_date = daystodate(days(),date_format);

days_of_july4_1993 = days('07041993','MMDDYYYY');
sep2_1993 = daystodate(days_of_july4_1993 + 60, Date_format);
             /*  09021993    */

date_due = daystodate(days() + 60, date_format);
        /* assuming today is July 4, 1993, this would be Sept. 2, 1993

msg = 'Please pay amount due on or before ' ||
         substr(date_due, 1, 2) || '/' ||
         substr(date_due, 3,2)  || '/' ||
         substr(date_due, 5);
```

DAYSTODATE

DAYSTODATE returns a nonvarying character string containing the date in the form *p* that corresponds to *d* days (in Lilian format).



d The number of days (in Lilian format).

d must have a computational type and is converted to FIXED BINARY(31,0) if necessary.

p One of the supported date patterns shown in Table 37 on page 515. If omitted, the compiler assumes that *p* is the default pattern returned by the DATETIME built-in function (YYYYMMDDHHMISS999).

p should have character type. If not, it is converted to character.

w An integer expression that defines a century window to be used to handle any two-digit year formats.

- If the value is positive, such as 1950, it is treated as a year.
- If negative or zero, the value specifies an offset to be subtracted from the current, system-supplied year.
- If omitted, *w* defaults to the value specified in the WINDOW compile-time option.

Performing date calculations and comparisons

Once you understand what the PL/I millennium language features are and you have made the appropriate syntax changes, you can use MLE to perform calculations and comparisons in your applications.

Explicit date calculations

You can use the DAYS and DAYSTODATE built-in functions to make date comparisons and calculations manually.

Comparing dates

To compare two dates `d1` and `d2` which have the date pattern `YYMMDD`, you can use the following code:

```
DAYS (d1, 'YYMMDD', w) < DAYS(d2, 'YYMMDD', w)
```

Converting dates

You can convert between a two-digit date (`d1`) with the pattern `YYMMDD` and a four-digit date (`d2`) with the pattern `YYYYMMDD` using assignments:

```
d2 = DAYSTODATE(DAYS(d1,'YYMMDD',w), 'YYYYMMDD');
d1 = DAYSTODATE(DAYS(d2,'YYYYMMDD'), 'YYMMDD' ,w);
```

Subtracting dates

To subtract 2 two-digit years, `y1` and `y2`, you need to calculate the imposing difference:

```
DAYSTODATE(DAYS(y1,'YY',w), 'YYYY') -
        DAYSTODATE(DAYS(y2,'YY',w),'YYYY')
```

Implicit date calculations

You can use MLE to take advantage of implicit date comparisons and conversions if you first complete the following steps:

- Give the two operands the DATE attribute
- Specify the RESPECT compile-time option

Implicit date comparisons

The DATE attribute causes implicit *commoning* when two variables declared with the DATE attribute are compared. Comparisons where only one variable has the DATE attribute are flagged, and the other comparand is generally treated as if it had the same DATE attribute, although some exceptions apply which are discussed later.

Implicit commoning means that the compiler generates code to convert the dates to a common, comparable representation. This process converts 2-digit years using the *window* you specify in the WINDOW compile-time option.

Implicit date comparisons

In the following code fragment, if the DATE attribute is honored, then the comparison in the second display statement is 'windowed'. This means that if the window started at 1900, the comparison would return false. However, if the window started at 1950, the comparison would return true.

```
dcl a   pic'(6)9' date;
dcl b   pic'(6)9' def(a);
dcl c   pic'(6)9' date;
dcl d   pic'(6)9' def(c);

b = '670101';
d = '010101';

display( b || ' < ' || d || ' ?' );
display( a < c );
```

Date comparisons can also occur in the following places:

- IF and SELECT statements
- WHILE or UNTIL clauses
- Implicit comparisons caused by a TO clause.

Comparing dates with like patterns

The compiler does not generate any special code to compare dates with identical patterns under the following conditions:

- The comparison operator of = or ¬= is used
- The pattern is equal to YYYY, YYYYMM, YYYYDDD, or YYYYMMDD.

Comparing dates with differing patterns

For comparisons involving dates with unlike patterns, the compiler generates code to convert the dates to a common comparable representation. Once the conversion has taken place, the compiler compares the two values.

Comparisons involving the DATE attribute and a literal

If you are making comparisons in which one comparand has the DATE attribute and the other is a literal, the compiler issues a W-level message. Further compiler action depends on the value of the literal as follows:

- If the literal appears to be a valid date, it is treated as if it had the same date pattern and window as the comparand with the DATE attribute.
- If the literal does not appear to be a valid date, the DATE attribute is ignored on the other comparand.

```
dcl start_date char(6) date;
if start_date >= '' then /* no windowing */
...
if start_date >= '851003' then /* windowed */
...
```

Comparisons involving the DATE attribute and a non-literal

In comparisons where one comparand has the DATE attribute and the other is not a date and not a literal, the compiler issues an E-level message. The non-date value is treated as if it had the same date pattern as the other comparand and as if it had the same window.

```
dcl start_date char(6) date;
dcl non_date char (6);

if start_date >= non_date then /* windowed */
...
```

Implicit DATE assignments

The DATE attribute can also cause implicit conversions to occur in assignments of two variables declared with date patterns.

- If the source and target have the same DATE and data attributes, then the assignment proceeds as if neither had the DATE attribute.
- If the source and target have differing DATE attributes, then the compiler generates code to convert the source date before making the assignment.
- In assignments where the source has the DATE attribute but the target does not, the compiler issues an E-level message and ignores the DATE attribute.
- In assignments where the target has the DATE attribute but the source does not (and the source IS NOT a literal), the compiler issues an E-level message and ignores the DATE attribute.
- In assignments where the target has the DATE attribute but the source does not (and the source IS a literal), the compiler issues a W-level message and ignores the DATE attribute.

  ```
  dcl start_date char(6) date;
  start_date = '';
  ...
  ```

- If the source holds a four-digit year and the target holds a two-digit year, the source can hold a year that is not in the target window. In this case, the ERROR condition is raised.

  ```
  dcl x char(6) date;
  dcl y char(8) date('YYYYMMDD');

  y = '20600101';

  x = y; /* raises error if window is <= 1960 */
  ```

- The DATE attribute is ignored in:
  - The debugger
  - Assignments performed in record I/O statements
  - Assignments and conversions performed in stream I/O statements (such as GET DATA).

Summarizing date diagnostics

Even if you do not choose a windowing solution, you might have some code that needs to manipulate both two- and four-digit years. You can use multiple date patterns to help you in these situations:

```
dcl old_date char(6) date('YYMMDD');
dcl new_date char(8) date('YYYYMMDD');

new_date = old_date;
```

Summarizing date diagnostics

In PL/I, *effective* assignments occur when

- An expression is passed as an argument to an entry that has described that argument
- An expression is used in a RETURN statement.

The following uses of variables with the DATE attribute are flagged:

- Assignments (explicit or effective) which include either
  - A date to a non-date
  - A non-date to a date
- Any arithmetic operation applied to a date
- Use of a date in a BY clause (since this implies an arithmetic operation)
- Use of a date in any mathematical built-in function
- Use of a date in any arithmetic built-in function except BINARY, DECIMAL, FIXED, FLOAT, or PRECISION
- Use of a date in the built-in functions SUM, PROD, or POLY.

In all of these cases, code is produced but no windowing occurs. In effect, the DATE attribute is ignored.

Using MLE with the SQL preprocessor

The SQL preprocessor objects to the DATE attribute. However, if you enclose the attribute between /* and */, the SQL preprocessor ignores it (as part of a comment that stretches from the first /* to the last */). In order for the compiler to honor the DATE attribute between these special characters, you must specify RULES(LAXCOMMENT), see “RULES compile-time option” on page 515 for more details.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74/G4
555 Bailey Avenue
San Jose, CA 95141-1099
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations might not appear.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM PL/I for MVS & VM.

Macros for customer use: IBM PL/I for MVS & VM provides no macros that allow a customer installation to write programs that use the services of IBM PL/I for MVS & VM.

Attention: Do not use as programming interfaces any IBM PL/I for MVS & VM macros.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
CICS
CICS/ESA
DFSMS/MVS
DFSORT
IBM
IMS
IMS/ESA
Language Environment
OS/2
OS/390
Proprinter
VisualAge

Windows is a trademark of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Bibliography

VisualAge PL/I publications

Fact Sheet, GC26-9470
Programming Guide, SC26-9473
Language Reference, SC26-9476
Messages and Codes, SC26-9478
Diagnosis Guide, SC26-9475
Compiler and Run-Time Migration Guide, SC26-9474
Installation and Customization, GC26-9472
Building Graphical User Interfaces on OS/2, GC26-9180-01

DB2 Version 2

Information and Concepts Guide, S20H-4664

Administration Guide, S20H-4580

Database System Monitor Guide and Reference, S20H-4871

Command Reference, S20H-4645

API Reference, S20H-4984

SQL Reference, S20H-4665

Application Programming Guide, S20H-4643

Call Level Interface Guide and Reference, S20H-4644

Messages Reference, S20H-4808

Problem Determination Guide, S20H-4779

DDCS User's Guide, S20H-4793

DRDA Connectivity Guide, SC26-4783

DATABASE 2

Application Programming and SQL Guide, SC26-4377

SQL Reference, SC26-4380

VisualAge CICS Enterprise Application Development

Installation, GC34-5356

Customization, SC34-5357

Operation, SC34-5358

Reference Summary, SX33-6109

Intercommunication, SC34-5359

Problem Determination, GC34-5360

Performance, SC34-5363

Application Programming, SC34-5361

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. (1) The state of a block after activation and before termination. (2) 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. (3) The state in which an event variable is said to be during the time it is associated with an asynchronous operation. (4) The state in which a task variable is said to be when its associated task is attached. (5) 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. (1) The reservation of main storage for a variable. (2) A generation of an allocated variable. (3) 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. (1) The overlap of an input/output operation with the execution of statements. (2) 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. (1) A descriptive property associated with a name to describe a characteristic represented. (2) 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. (1) A 0 or a 1. (2) 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. (1) A series of binary digits enclosed in and followed immediately by the suffix B. Contrast with *character constant*. (2) 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. (1) An arithmetic or string data item that does not have a name and whose value cannot change. (2) An identifier declared with the VALUE attribute. (3) 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. (1) A collection of data external to the program that can be accessed by reference to a single file name. (2) 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. (1) The establishment of an identifier as a name and the specification of a set of attributes (partial or complete) for it. (2) 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. (1) The label prefix of a PROCEDURE statement (an entry name). (2) 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. (1) An identifier that is explicitly or contextually declared to have the ENTRY attribute (unless the VARIABLE attribute is given) or (2) 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. (1) A notation, within a program, that represents a value, an array of values, or a structured set of values. (2) 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. (1) The range indicated by the bounds of an array dimension, by the length of a string, or by the size of an area. (2) 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. (1) A procedure that is not contained in any other procedure. (2) 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). (1) A procedure that has a RETURNS option in the PROCEDURE statement. (2) 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. (1) An optionally signed sequence of digits or a sequence of bits without a decimal or binary point. (2) 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. (1) 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. (2) 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. (1) 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. (2) 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. (1) A major structure or union name. (2) 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. (1) A structure, union, or element name in a structure or union. (2) 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. (1) Two or more declarations of the same identifier internal to the same block without different qualifications. (2) 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. (1) One or more characters, graphics, or bits concatenated to the right of a string to extend the string to a required length. (2) 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. (1) The logical unit of transmission in a record-oriented input or output operation. (2) 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

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

X

We'd Like to Hear from You

VisualAge PL/I
Programming Guide
Version 2.1

Publication No. GC26-9177-01

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:

  Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:

  – Title and publication number of this book
  – Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

VisualAge PL/I
Programming Guide
Version 2.1

Publication No. GC26-9177-01

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

May we contact you to discuss your comments? Yes No

Would you like to receive our response by E-Mail?

_____

Your E-mail address

_____ _____
Name Address

_____ _____
Company or Organization

_____ _____
Phone No.

Readers' Comments
GC26-9177-01

IBM®

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department HHX/H1
San Jose, CA 95141-1099

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

IBM®

Program Number: 5639-D65

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

Common VisualAge PL/I Library

SC26-9476	Language Reference

VisualAge PL/I for OS/390 Library

GC26-9471	Licensed Program Specifications
SC26-9473	Programming Guide
SC26-9474	Compiler and Run-Time Migration Guide
SC26-9475	Diagnosis Guide
SC26-9478	Compile-Time Messages and Codes

VisualAge PL/I Enterprise Version 2.1 Library

GC26-9178	Language Reference (OS/2 and Windows)
GC26-9177	Programming Guide (OS/2 and Windows)
GC26-9179	Messages and Codes (OS/2 and Windows)
GC26-9180	Building Graphical User Interfaces on OS/2

GC26-9177-01

Spine information:

IBM VisualAge PL/I Programming Guide *Version 2.1*