IBM VisualAge PL/I for OS/390

Programming Guide

Version 2 Release 2.1

SC26-9473-01

IBM VisualAge PL/I for OS/390

Programming Guide

Version 2 Release 2.1

SC26-9473-01

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix, “Notices” on page 256.

Second Edition (September 2000)

This edition applies to Version 2 Release 2.1 of IBM® VisualAge® PL/I for OS/390®, 5655-B22, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

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

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

IBM Corporation, Department HHX/H3
P.O. Box 49023
San Jose, CA, 95161-9023
United States of America

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

Contents

Part 1. Introduction

About This Book

This book is for PL/I programmers and system programmers. It helps you understand how to use VisualAge PL/I for OS/390 in order to compile PL/I programs. It also describes the operating system features that you might need to optimize program performance or handle errors.

Important: VisualAge PL/I for OS/390 will be referred to as VisualAge PL/I throughout this book.

Run-time environment for VisualAge PL/I for OS/390

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

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

Using your documentation

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

The following tables show you how to use the publications you receive with VisualAge PL/I and Language Environment. You'll want to know information about both your compiler and run-time environment. For the complete titles and order numbers of these and other related publications, see “Bibliography” on page 259.

PL/I information

Table 1. How to use VisualAge PL/I publications

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

Language Environment information

Table 2. How to use OS/390 Language Environment publications

To...	Use...
Evaluate Language Environment	Concepts Guide
Plan for Language Environment	Concepts Guide Run-Time Migration Guide
Install Language Environment on OS/390	OS/390 Program Directory
Customize Language Environment on OS/390	Customization
Understand Language Environment program models and concepts	Concepts Guide Programming Guide
Find syntax for Language Environment run-time options and callable services	Programming Reference
Develop applications that run with Language Environment	Programming Guide and your language Programming Guide
Debug applications that run with Language Environment, get details on run-time messages, diagnose problems with Language Environment	Debugging Guide and Run-Time Messages
Develop interlanguage communication (ILC) applications	Writing Interlanguage Applications
Migrate applications to Language Environment	Run-Time Migration Guide and the migration guide for each Language Environment-enabled language

Notation conventions used in this book

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

Conventions used

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

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

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

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

How to read the syntax notation

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

Arrow symbols

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

▶▶──	Indicates the beginning of a statement.
──▶	Indicates that the statement syntax is continued on the next line.
▶──	Indicates that a statement is continued from the previous line.
──▶◀	Indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the ▶── symbol and end with the ──▶ symbol.

Conventions

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

Required items

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

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

Optional Items

Optional items appear below the main path.

```
►►──REQUIRED_ITEM──┬──────────────┬──────────────────────►◄
                   └─optional_item─┘
```

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

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

Multiple required or optional items

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

```
►►──REQUIRED_ITEM──┬─required_choice1─┬──────────────────►◄
                   └─required_choice2─┘
```

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

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

Repeatable items

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

```
                   ┌───────────────┐
►►──REQUIRED_ITEM──▼─repeatable_item─┴───────────────────►◄
```

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

```
                   ┌─,─────────────┐
►►──REQUIRED_ITEM──▼─repeatable_item─┴───────────────────►◄
```

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

Default keywords

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

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

Fragments

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

```
►►──STATEMENT──item 1──item 2──┤ A ├────────────────────────────────►◄
```

A:

```
├──┬─item 3─┬──KEYWORD──┬─item 5─┬──────────────────────────────────┤
   └─item 4─┘           └─item 6─┘
```

Substitution-block

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

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

where <A> is:

```
►►──┬─EITHER─┬───────────────────────────────────────────────────────►◄
    └─OR─────┘
```

Parameter endings

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

```
►►──/MSVERIFY──┬─MSNAME──msname─┬────────────────────────────────────►◄
               └─SYSID──sysid#──┘
```

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

How to read the notational symbols

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

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

- An **ellipsis**, ..., indicates that multiple entries of the type immediately preceding the ellipsis are allowed.

Example of notation

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

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

Interpret this example as follows:

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

Part 2. Compiling your program

Chapter 1. Using compile-time options and facilities

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

Compile-time option descriptions

There are three types of compile-time options; however, most compile-time options have a positive and negative form. The negative form is the positive with 'NO' added at the beginning (as in TEST and NOTEST). Some options have only a positive form (as in SYSTEM). The three types of compile-time options are:

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

Table 3 lists all compile-time options with their abbreviated syntax and their IBM-supplied default values.

The paragraphs following Table 3 describe the options in alphabetical order. For those options specifying that the compiler is to list information, only a brief description is included; the generated listing is described under "Using the compiler listing" on page 41.

Table 3 (Page 1 of 2). Compile-time options, abbreviations, and IBM-supplied defaults

Compile-Time Option	Abbreviated Name	OS/390 Default
AGGREGATE ∣ NOAGGREGATE	AG ∣ NAG	NAG
ARCH	–	ARCH(0)
ATTRIBUTES[(FULL∣SHORT)] ∣ NOATTRIBUTES	A[(F ∣ S)] ∣ NA	NA [(FULL)]1
CHECK(STORAGE ∣ NOSTORAGE)	CHECK(STG ∣ NSTG)	CHECK(NSTG)
COMPILE ∣ NOCOMPILE[(W ∣ E ∣ S)]	C ∣ NC[(W ∣ E ∣ S)]	NC(S)
CURRENCY(x)	–	CURRENCY($)
DD(ddname-list)	–	DD(SYSPRINT,SYSIN,SYSLIB, SYSPUNCH,SYSLIN)
DEFAULT(*attribute ∣ option*)	DFT	See page 14
DISPLAY(STD ∣ WTO)	–	DISPLAY(WTO)
DLLINIT ∣ NODLLINIT	–	NODLLINIT
EXIT ∣ NOEXIT	–	NOEXIT
EXTRN(FULL ∣ SHORT)	EXTRN(F ∣ S)	EXTRN(SHORT)
FLAG[(I ∣ W ∣ E ∣ S ∣ n)]	F[(I ∣ W ∣ E ∣ S ∣ n)]	F(W)
FLOAT(AFP ∣ NOAF)	–	FLOAT(NOAFP)
GONUMBER ∣ NOGONUMBER	GN ∣ NGN	NGN
GRAPHIC ∣ NOGRAPHIC	GR ∣ NGR	NGR
INCAFTER([PROCESS(*filename*)])	–	INCAFTER()
INCDIR('*directory name*')	–	INCDIR()

Table 3 (Page 2 of 2). Compile-time options, abbreviations, and IBM-supplied defaults

Compile-Time Option	**Abbreviated Name**	**OS/390 Default**
INCLUDE | NOINCLUDE	INC | NINC	NINC
INSOURCE | NOINSOURCE	IS | NIS	IS
INTERRUPT | NOINTERRUPT	INT | NINT	NINT
LANGLVL({SAA | SAA2[,NOEXT | OS])	–	LANGLVL(SAA2,OS)
LIMITS(*options*)	–	See page 19
LINECOUNT(*n*)	LC	LC(60)
LIST | NOLIST	–	NOLIST
MACRO | NOMACRO	M | NM	NM
MAP | NOMAP	–	NOMAP
MARGINI('c') | NOMARGINI	MI('c') | NMI	NMI
MARGINS(m,n)	MAR(m,n)	MAR F-format: (2,72) V-format: (10,100)
MAXMEM(n)	–	MAXMEM(1048576)
MDECK | NOMDECK	MD | NMD	NMD
NAMES	–	NAMES('#@$' '#@$')
NEST | NONEST	–	NONEST
NOT	–	NOT('¬')
OBJECT | NOOBJECT	OBJ | NOBJ	OBJ
OFFSET | NOOFFSET	OF | NOF	NOOFFSET
OPTIMIZE(TIME | 0 | 2) | NOOPTIMIZE	OPT(TIME|0|2)|NOPT	NOPT
OPTIONS | NOOPTIONS	OP | NOP	NOP
OR('c')	–	OR(' | ')
PP(*pp-name*) | NOPP	–	NOPP
PPTRACE | NOPPTRACE	–	NOPPTRACE
PREFIX(*condition*)	–	See page 26
PROCEED | NOPROCEED(S | E | W)	PRO | NPRO	NPRO(S)
RESPECT([DATE])	–	RESPECT()
RULES(*options*)	LAXCOM | NOLAXCOM	See page 27
SEMANTIC | NOSEMANTIC(S | E | W)	SEM | NSEM	SEM or NSEM(S)
SOURCE | NOSOURCE	S | NS	NS
STORAGE | NOSTORAGE	STG | NSTG	NSTG
SYNTAX | NOSYNTAX(W | E | S)	SYN | NSYN(W | E | S)	NSYN(S)
SYSPARM('string')	–	SYSPARM('')
SYSTEM(MVS | CICS | IMS)	–	MVS
TERMINAL | NOTERMINAL	TERM | NTERM	NTERM
TEST(ALL | NONE | STMT,SYM | ,NOSYM) | NOTEST	–	NOTEST(NONE,SYM)[2]
SPILL(n)	–	SPILL(512)
TUNE(n)	–	TUNE(0)
WIDECHAR(BIGENDIAN | LITTLEENDIAN)	–	WIDECHAR(BIGENDIAN)
WINDOW(w)	–	WINDOW(1950)
XREF[(FULL | SHORT)] | NOXREF	X[(F | S)] | NX	NX [(FULL)][1]

Notes:

1. FULL is the default suboption if the suboption is omitted with ATTRIBUTES or XREF.
2. (NONE,SYM) is the default suboption if the suboption is omitted with TEST.

AGGREGATE

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

```
     ┌─NOAGGREGATE─┐
►►───┴─AGGREGATE───┴──────────────────────────────────────────────────►◄
```

ABBREVIATIONS: AG, NOAG

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

ARCH

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

```
                ┌─0─┐
►►──ARCH──(─────┴─n─┴───)─────────────────────────────────────────────►◄
```

Current groups of models that are supported include the following:

0 Produces code that is executable on all models.

1 Produces code that is optimized for the following models:

> 9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900
>
> 9021-xx1 and 9021-xx2
>
> 9672-Rx1, 9672-Exx, and 9672-Pxx

2 Produces code that is optimized for the following and follow-on models:

> 9672-Rx2, 9672-Rx3, 9672-Rx4, and 2003

3 Produces code that is optimized for the 9672 G5 and follow-on models

Note: Code that is compiled at ARCH(1) runs on machines in the ARCH(1) group and later machines, including those in the ARCH(2) group. It cannot run on earlier machines. Code that is compiled at ARCH(2) cannot run on ARCH(1) or earlier machines.

ATTRIBUTES

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

```
     ┌─NOATTRIBUTES─┐
►►───┴─ATTRIBUTES───┴──┬─────────────────────┬────────────────────────►◄
                       │      ┌─FULL──┐      │
                       └─(────┴─SHORT─┴──)───┘
```

ABBREVIATIONS: A, NA, F, S

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

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

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

CHECK

The CHECK option alters the behavior of the ALLOCATE and FREE statements.

```
                ┌─NOSTORAGE─┐
►►──CHECK──(────┴─STORAGE───┴──)───────────────────────────────────────►◄
```

ABBREVIATIONS: STG, NSTG

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

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

COMPILE

The COMPILE option causes the compiler to stop compiling after all semantic checking of the source program if it produces a message of a specified severity during preprocessing or semantic checking. Whether the compiler continues or not depends on the severity of the error detected, as specified by the NOCOMPILE option in the list below. The NOCOMPILE option specifies that processing stops unconditionally after semantic checking.

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

ABBREVIATIONS: C, NC

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

NOCOMPILE
Compilation stops after semantic checking.

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

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

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

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

CURRENCY

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

```
                    ┌─$─┐
►►──CURRENCY──(──'──┴─x─┴──'──)──────────────────────────────────────►◄
```

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

DD

The DD option allows you to specify alternate DD names for the compiler listing, the primary source file, the default include dataset and the mdeck dataset.

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

Up to five DD names may be specified. In order, they specify alternate DD names for

- SYSPRINT
- SYSIN
- SYSLIB
- SYSPUNCH
- SYSLIN

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

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

DEFAULT

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

```
►►──DEFAULT──(──┬──────────────────────────────────┬──)────────────────────────►◄
                │  ┌─,──────────────────────────┐  │
                │  ↓ ┌─IBM─┐                    │  │
                └────┴─ANS─┴────────────────────┴──┘
                   ├─┬─EBCDEC─┬──────────────────┤
                   │ └─ASCII──┘                  │
                   ├─┬─ASSIGNABLE────┬───────────┤
                   │ └─NONASSIGNABLE─┘           │
                   ├─┬─BYADDR──┬─────────────────┤
                   │ └─BYVALUE─┘                 │
                   ├─┬─NONCONNECTED─┬────────────┤
                   │ └─CONNECTED────┘            │
                   ├─┬─DESCRIPTOR───┬────────────┤
                   │ └─NODESCRIPTOR─┘            │
                   ├─┬─NATIVE────┬───────────────┤
                   │ └─NONNATIVE─┘               │
                   ├─┬─NATIVEADDR────┬───────────┤
                   │ └─NONNATIVEADDR─┘           │
                   ├─┬─NOINLINE─┬────────────────┤
                   │ └─INLINE───┘                │
                   ├─┬─ORDER───┬─────────────────┤
                   │ └─REORDER─┘                 │
                   │                ┌─OPTLINK─┐  │
                   ├─LINKAGE──(─────┴─SYSTEM──┴──)─┤
                   ├─┬─EVENDEC───┬───────────────┤
                   │ └─NOEVENDEC─┘               │
                   ├─┬─NULL370─┬─────────────────┤
                   │ └─NULLSYS─┘                 │
                   ├─┬─NONRECURSIVE─┬────────────┤
                   │ └─RECURSIVE────┘            │
                   ├─┬─DESCLOCATOR─┬─────────────┤
                   │ └─DESCLIST────┘             │
                   │                ┌─BYADDR──┐  │
                   ├─RETURNS──(─────┴─BYVALUE─┴──)─┤
                   │             ┌─HEXADEC─┐     │
                   ├─SHORT──(────┴─IEEE────┴──)──┤
                   │             ┌─ALIGNED───┐   │
                   ├─DUMMY──(────┴─UNALIGNED─┴──)─┤
                   ├─┬─LOWERINC─(1)─┬────────────┤
                   │ └─UPPERINC─────┘            │
                   ├─┬─NORETCODE─┬───────────────┤
                   │ └─RETCODE───┘               │
                   ├─┬─ALIGNED───┬───────────────┤
                   │ └─UNALIGNED─┘               │
                   │                ┌─MIN─┐      │
                   ├─ORDINAL──(─────┴─MAX─┴──)───┤
                   │ ┌─NOOVERLAP─┐               │
                   └─┴─OVERLAP───┴───────────────┘
```

Note:
1 For OS/390 UNIX only.

ABBREVIATIONS: DFT, ASGN, NONASGN, NONCONN, CONN

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

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

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

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

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

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

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

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

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

NATIVE | NONNATIVE

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

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

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

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

Other combinations produce unpredictable results.

NATIVEADDR | NONNATIVEADDR

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

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

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

Other combinations produce unpredictable results.

INLINE | NOINLINE

This option sets the default for the inline procedure option.

Specifying INLINE allows your code to run faster but, in some cases, also creates a larger executable file. For more information on how inlining can improve the performance of your application, see Chapter 11, "Improving performance" on page 204.

ORDER | REORDER

Affects optimization of the source code. Specifying REORDER allows further optimization of your source code, see Chapter 11, "Improving performance" on page 204.

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 system APIs.

EVENDEC | NOEVENDEC

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

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

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

EVENDEC is the default.

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

NULL370 is the default.

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

NONRECURSIVE is the default.

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

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

DESCLOCATOR is the default.

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

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

SHORT (HEXADEC) is the default.

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

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

Consider the following example:

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

dcl x entry( fixed bin(31) );

call x( b3 );
```

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

DUMMY(ALIGNED) is the default.

LOWERINC | UPPERINC (for OS/390 UNIX only)
If you specify LOWERINC, the compiler requires that the names of INCLUDE files are in lowercase. If you specify UPPERINC, the compiler requires that the names are in uppercase.

LOWERINC is the default.

RETCODE | 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.

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

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

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

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

ALIGNED is the default.

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

ORDINAL(MIN) is the default.

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

NOOVERLAP is the default.

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

DISPLAY

The DISPLAY option determines where out of the DISPLAY statement is directed.

```
                 ┌─WTO─┐
►►──DISPLAY──(───┴─STD─┴───)──────────────────────────────────────────►◄
```

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

WTO
All DISPLAY statements are completed via WTOs. This is the default.

DLLINIT

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

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

NODLLINIT has no affect on your programs.

EXIT

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

```
     ┌─NOEXIT──────────────────────────────┐
►►───┴─EXIT───┬───────────────────────────┬┴──────────────────────────►◄
              └─(────inparm_string────)───┘
```

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

EXTRN

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

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

FULL
EXTRNs are emitted for all declared external entry constants.

SHORT
EXTRNs are emitted only for those constants that are referenced. This is the default.

FLAG

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

```
►►──FLAG──┬────────────────────────────┬──────────────────────────────────►◄
          │      ┌──────────────────┐  │
          │      │   ┌─,─┐          │  │
          │      ▼   │   │          │  │
          └─(────┴─┬─┼─W─┼──────────┴─)┘
                   │ ├─I─┤        │
                   │ ├─E─┤        │
                   │ └─S─┘        │
                   │ ┌─250─┐      │
                   └─┴─n───┴──────┘
```

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.

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.

FLOAT

The FLOAT option controls the use of additional floating-point registers.

```
                 ┌─NOAFP─┐
►►──FLOAT──(─────┴─AFP───┴──)──────────────────────────────────────────────►◄
```

FLOAT(NOAFP)
Compiler-generated code uses the traditional 4 floating-point registers.

FLOAT(AFP)
Compiler-generated code uses 16 floating-point registers.

GONUMBER

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

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

ABBREVIATIONS: GN, NGN

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

GONUMBER is forced by the ALL and STMT suboptions of the TEST option. The OFFSET option is separate from these numbering options and must be specified if required.

GRAPHIC

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

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

ABBREVIATIONS: GR, NGR

The GRAPHIC option must be specified if the source program uses any of the following:

- DBCS identifiers
- Graphic string constants
- Mixed-string constants
- Shift codes anywhere else in the source

INCAFTER

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

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

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

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

Consider the following example:

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

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

INCDIR

The INCDIR compile-time option includes a directory in the search path for the location of include files.

```
►►──INCDIR──(──'directory name'──)──────────────────────────────────►◄
```

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

The compiler looks for INCLUDE files in the following order:

1. Current directory
2. Directories specified with the –I flag or with the INCDIR compile-time option
3. /usr/include directory

INCLUDE

The INCLUDE 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 specified in the INCDIR option.

Note: This option does not apply to the batch jobs.

```
►►──INCLUDE──(─┬─────────────────────────────────────┬─)────────────────────►◄
               │         ┌──────────────────────┐    │
               │         │ └─,─┘                │    │
               │         ▼  ┌─inc─┐             │    │
               └─EXT──(─────┴─────┴──ext_string─┴──)─┘
```

ABBREVIATION: INC

The extension string can be up to 31 characters long, but it is truncated to the first three characters. 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 under DFT(UPPERINC), to lowercase under DFT(LOWERINC).

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 that you specified from left to right. You can specify a maximum of 7 extensions.

Do not use `PLI` as an extension for an include file.

INSOURCE

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

```
    ┌─NOINSOURCE─┐
►►──┴─INSOURCE───┴──────────────────────────────────────────────────►◄
```

ABBREVIATION: IS, NIS

The INSOURCE listing contains preprocessor statements that do not appear in the SOURCE listing. This option is applicable only when the MACRO option is in effect.

INTERRUPT

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

```
      ┌─NOINTERRUPT─┐
►►────┴─INTERRUPT───┴──────────────────────────────────────────────►◄
```

ABBREVIATION: INT, NINT

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

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

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

If you require the attention interrupt capability only for testing purposes, use the TEST option instead of the INTERRUPT option. For more information see “TEST” on page 33.

See Chapter 15, “Interrupts and attention processing” on page 244 for more information about using interrupts in your programs.

LANGLVL

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

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

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

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

NOEXT
: No extensions beyond the language level specified are allowed.

OS
: The ENVIRONMENT options are allowed, such as Variable Unblocked (V) and Variable Blocked (VB). For a complete list of the ENVIRONMENT options, see Table 10 on page 108.

LIMITS

The LIMITS option specifies various implementation limits.

```
►►──LIMITS──(───────────────────────────────────────────────────────────►

   ┌─────────────────────────────────────────────────────────────┐
   │ └─,─┘                                                       │
   │               ┌─100─┐                          ┌─100─┐     │
►──▼─┬─EXTNAME──(──┴─n───┴──)───────────┬─NAME──(──┴─n───┴──)──┴──)──────►◄
     │              ┌─15─┐              │
     ├─FIXEDDEC──(──┴─31─┴──)───────────┤
     │              ┌─31─┐              │
     └─FIXEDBIN──(──┴─63─┴─┬──────────┬─)─┘
                           │    ┌─31─┐│
                           └─,──┴─63─┴┘
```

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.

LINECOUNT

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

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

ABBREVIATION: LC

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

LIST

The LIST option provides a listing of the object module (in a syntax similar to assembler language instructions) in the compiler listing.

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

MACRO

The MACRO option invokes the preprocessor.

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

MAP

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

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

The MAP option forces the LIST option.

MARGINI

The MARGINI option provides a specified character in the column preceding the left-hand margin, and also in the column following the right-hand margin, of the listings produced by the INSOURCE and SOURCE options. The compiler shifts any text in the source input that precedes the left-hand margin left one column. It shifts any text that follows the right-hand margin right one column. Thus you can easily detect text outside the source margins.

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

ABBREVIATIONS: MI, NMI

c The character to be printed as the margin indicator.

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

MARGINS

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

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

```
              ┌─2─┐     ┌─72─┐
►►──MARGINS──(──┴─m─┴──,──┴─n──┴──┬───────┬──)──────────────────────►◄
                                  └─,──c──┘
```

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 100.

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

c The column number of the ANS printer control character. It must not exceed 100 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 *PL/I Language Reference*).

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

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

MAXMEM

When compiling with OPTIMIZE, the MAXMEM option limits the amount of memory used for local tables of specific, memory intensive optimizations to the specified number of kilobytes. The maximum number of kilobytes that may be specified is 1048576, which is also the default. Use the MAXMEM option if you want to specify a memory size of less value than the default.

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

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

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

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

MDECK

The MDECK option specifies that the preprocessor produces a copy of its output either on the file defined by the SYSPUNCH DD statement under OS/390, or on the .dek file under OS/390 UNIX.

```
     ┌─NOMDECK─┐
►►───┴─MDECK───┴────────────────────────────────────────────────►◄
```

ABBREVIATIONS: MD, NMD

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

NAMES

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

```
                  ┌───────────────┐
►►──NAMES──(──'──▼extraling_char─┴─'──┬──────────────────────────────────┬──)──────►◄
                                      │          ┌────────────────────┐  │
                                      └─┬───┬─'─▼upp_extraling_char─┴─'┘
                                        └─,─┘
```

extracting_char
   An extralingual character

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

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

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

NEST

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

```
     ┌─NONEST─┐
►►───┴─NEST───┴──────────────────────────────────────────────────────►◄
```

NOT

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

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

char
A single SBCS character.

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

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

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

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

OBJECT

The OBJECT option specifies that the compiler either creates an object module and stores it in a data set defined by the DD statement with the name SYSLIN under OS/390, or creates a .o file under OS/390 UNIX.

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

ABBREVIATIONS: OBJ, NOBJ

OFFSET

The OFFSET option controls whether the offsets shown in the assembler listing are from the start of the current module or the current procedure.

```
      ┌─NOOFFSET─┐
      ├─NOF──────┤
►►────┼─OFFSET───┼──────────────────────────────────────────────────────────►◄
      └─OF───────┘
```

OPTIMIZE

The OPTIMIZE option specifies the type of optimization required:

```
      ┌─NOOPTIMIZE─┐
►►────┴─OPTIMIZE───┴──(───┬─TIME─┬──)───────────────────────────────────────►◄
                          ├─0────┤
                          └─2────┘
```

ABBREVIATIONS: OPT, NOPT

OPTIMIZE(TIME)
Optimizes the machine instructions generated to produce a more efficient object program. This type of optimization can also reduce the amount of main storage required for the object module. The use of OPTIMIZE(TIME) could result in a substantial increase in compile time over NOOPTIMIZE and a substantial increase in the space required. For example, compiling an average size program at OPT(TIME) might take several CPU minutes and could require a region of 50M or more. During optimization the compiler can move code to increase run-time efficiency. As a result, statement numbers in the program listing might not correspond to the statement numbers used in run-time messages.

OPTIMIZE(0)
The equivalent of NOOPTIMIZE.

OPTIMIZE(2)
The equivalent of OPTIMIZE(TIME).

NOOPTIMIZE
Specifies fast compilation speed, but inhibits optimization.

OPTIONS

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

```
      ┌─NOOPTIONS─┐
►►────┴─OPTIONS───┴─────────────────────────────────────────────────────────►◄
```

ABBREVIATIONS: OP, NOP

This list includes all options applied by default, those specified in the PARM parameter of an EXEC statement or in the invoking command (pli), those specified in a %PROCESS statement, those specified in the IBM_OPTIONS environment variable under OS/390, and all those incorporated from any options file.

OR

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

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

Note: Do not code any blanks between the quotes.

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

char
A single SBCS character.

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

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

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

PP

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

```
     ┌─NOPP──────────────────────────────────────────────┐
     │          ┌─,─┐                                    │
     │        ┌─┴───┴────────────────────────────────┐   │
►►───┴─PP──(──▼─pp-name──┬─────────────────────────┬─┴─)─┴──────────────►◄
                         │          ┌─,─┐          │
                         │        ┌─┴───┴──────┐   │
                         └─(──────▼─pp-string──┴─)─┘
```

pp-name
The name given to a particular preprocessor. INCLUDE and MACRO 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.

You can specify a maximum of 31 preprocessors.

PPTRACE

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

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

PREFIX

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

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

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

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

PROCEED

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

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

ABBREVIATIONS: PRO, NPRO

PROCEED
Is equivalent to NOPROCEED(S).

NOPROCEED
Ends the processing after the preprocessor has finished compiling.

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

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

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

RESPECT

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

```
►►──RESPECT──(──┬──────┬──)──────────────────────────────────────────►◄
                └─DATE─┘
```

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

RULES

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

```
              ┌────────────────────────┐
              │  └─,─┘                 │
              ▼   ┌─IBM─┐              │
►►──RULES──(──────┴─ANS─┴──────────────┴──)──────────────────────────►◄
                ├─┬─BYNAME────┬────────┤
                │ └─NOBYNAME──┘        │
                ├─┬─GOTO───┬───────────┤
                │ └─NOGOTO─┘           │
                ├─┬─NOLAXBIF─┬─────────┤
                │ └─LAXBIF───┘         │
                ├─┬─LAXCTL───┬─────────┤
                │ └─NOLAXCTL─┘         │
                ├─┬─NOLAXDCL─┬─────────┤
                │ └─LAXDCL───┘         │
                ├─┬─NOLAXIF─┬──────────┤
                │ └─LAXIF───┘          │
                ├─┬─LAXLINK───┬────────┤
                │ └─NOLAXLINK─┘        │
                ├─┬─LAXMARGINS───┬─────┤
                │ └─NOLAXMARGINS─┘     │
                ├─┬─LAXQUAL───┬────────┤
                │ └─NOLAXQUAL─┘        │
                ├─┬─NOLAXSTRZ─┬────────┤
                │ └─LAXSTRZ───┘        │
                ├─┬─NOLAXCOMMENT─┬─────┤
                │ └─LAXCOMMENT───┘     │
                └─┬─MULTICLOSE───┬─────┘
                  └─NOMULTICLOSE─┘
```

ABBREVIATIONS: LAXCOM, NOLAXCOM

IBM|ANS

Under the IBM suboption:

- For operations requiring string data, data with the BINARY attribute is converted to BIT.
- The second argument to the ROUND built-in function is ignored if the first argument has the FLOAT attribute.
- Conversions in arithmetic operations or comparisons occur as described in the *PL/I Language Reference*.
- Conversions for the ADD, DIVIDE, MULTIPLY, and SUBTRACT built-in functions occur as described in the *PL/I Language Reference* except that operations specified as scaled fixed binary are evaluated as scaled fixed decimal.
- Nonzero scale factors are permitted in FIXED BIN declares.
- If the result of any precision-handling built-in function (ADD, BINARY, etc.) has FIXED BIN attributes, the specified or implied scale factor can be nonzero.
- UNSPEC cannot be applied to a structure and, if applied to an array, returns an array of bit strings.
- Even if all arguments to the MAX or MIN built-in functions are UNSIGNED FIXED BIN, the result is always SIGNED.
- Even when you ADD, MULTIPLY, or DIVIDE two UNSIGNED FIXED BIN operands, the result has the SIGNED attribute.
- Even when you apply the MOD or REM built-in functions to two UNSIGNED FIXED BIN operands, the result has the SIGNED attribute.

Under the ANS suboption:

- For operations requiring string data, data with the BINARY attribute is converted to CHARACTER.
- The ROUND built-in function is implemented as described in the *PL/I Language Reference*.
- 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.
- UNSPEC can be applied to structures and, when applied to a structure or an array, UNSPEC returns a single bit string.

BYNAME|NOBYNAME

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

GOTO|NOGOTO
Specifying NOGOTO causes all GOTO statements to be flagged.

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

LAXCOMMENT|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.

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

The following is illegal under NOLAXCTL:

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

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

LAXIF|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|NOLAXLINK
Specifying LAXLINK causes the compiler to ignore the LINKAGE and other options specified in the declarations of two ENTRY variables or constants when you assign or compare one with the other.

LAXMARGINS|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.

LAXQUAL|NOLAXQUAL

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

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

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

LAXSTRZ|NOLAXSTRZ

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

MULTICLOSE|NOMULTICLOSE

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

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

SEMANTIC

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

```
    ┌─SEMANTIC──────────────────┐
►►──┴─NOSEMANTIC─┬──────────────┬┴──────────────────────────────────────►◄
                 └─(──┬─S─┬──)─┘
                      ├─E─┤
                      └─W─┘
```

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.

SPILL

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

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

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

SOURCE

The SOURCE option specifies that the compiler includes a listing of the source program in the compiler listing. The source program listed is either the original source input or, if the MACRO option applies, the output from the preprocessor.

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

ABBREVIATIONS: S, NS

STORAGE

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

```
     ┌─NOSTORAGE─────────────┐
►►───┴─STORAGE──┬───────────┬┴─────────────────────────────────────►◄
                └─(──max──)─┘
```

ABBREVIATIONS: STG, NSTG

max
The limit for the number of bytes that can be used for compiler-generated temporaries. The compiler flags any statement that uses more bytes than those specified by max. The default for max is 100.

SYNTAX

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

```
      ┌──NOSYNTAX─────────┬─────────────┬─┐
      │                   │    ┌─S─┐    │ │
      │                   └─(──┼─W─┼──)─┘ │
      │                        └─E─┘      │
►►────┴─SYNTAX────────────────────────────┴──────────────────────────────►◄
```

ABBREVIATIONS: SYN, NSYN

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

NOSYNTAX
Processing stops unconditionally after preprocessing.

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

NOSYNTAX(E)
No syntax checking if the compiler detects an error, severe error, or unrecoverable error.

NOSYNTAX(S)
No syntax checking if the compiler detects a severe error or unrecoverable error.

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

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

SYSPARM

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

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

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

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

SYSTEM

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

```
               ┌MVS──┐
►►──SYSTEM──(──┼CICS─┼──)──────────────────────────────────────────►◄
               └IMS──┘
```

OS/390, CICS, and IMS are the subparameters recognized. This option allows a program compiled under one system to run under another.

Table 4 shows the type of parameter list you can expect, and how the program runs under the specified host system. It also shows the implied settings of NOEXECOPS. Run-time information for the SYSTEM option is provided in *OS/390 Language Environment Programming Guide.*. .

Table 4. SYSTEM option table

SYSTEM option	**Type of parameter list**	**Program runs as**	**NOEXECOPS implied**
SYSTEM(MVS)	Single varying character string or no parameters.	OS/390 application program	NO
	Otherwise, arbitrary parameter list.		YES
SYSTEM(CICS)	Pointer(s)	CICS® transaction	YES
SYSTEM(IMS)	Pointer(s)	IMS™ application program	YES

TERMINAL

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

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

ABBREVIATIONS: TERM, NTERM

TERMINAL
Messages are displayed on the terminal.

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

TEST

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

```
    ┌─NOTEST──────────────────────────────────────────┐
►►──┴─TEST──┬────────────────────────────────────────┬┴──────────────────►◄
            └─(──┬─┬─NONE──┬──┬─────────────────┬──┬─)─┘
                 │ ├─BLOCK─┤  │   ┌─SYM───┐     │  │
                 │ ├─STMT──┤  └─,─┴─NOSYM─┴─────┘  │
                 │ ├─PATH──┤                       │
                 │ └─ALL───┘                       │
                 │ ┌─SYM───┐                       │
                 └─┴─NOSYM─┴───────────────────────┘
                              ┌──────────────────┐
                              │   ┌─NONE──┐      │
                              └─,─┼─BLOCK─┼──────┘
                                  ├─STMT──┤
                                  ├─PATH──┤
                                  └─ALL───┘
```

STMT
Inserts hooks at statement boundaries and block boundaries. STMT generates a statement table.

PATH
Tells the compiler to intert hooks:

- Before the first statement enclosed by an interative DO statement
- Before the first statement of the true part of an IF statement
- Before the first statement of the false part of an IF statement
- Before the first statement of a true WHEN or OTHERWISE statement of a SELECT group
- Before the statement following a user label
- At CALLs or function references - both before and after control is passed to the routine
- At block boundaries

When PATH is specified, the compiler generates a statement table.

BLOCK
Tells the compiler to insert hooks at block boundaries (block entry and block exit).

ALL
Inserts hooks at all possible locations and generates a statement table.

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

NONE
No hooks are put into the program.

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

NOSYM
No symbol table is generated.

NOTEST
Suppresses the generation of all testing information.

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

If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following:

- The INTERRUPT option
- A TEST option other than NOTEST or TEST(NONE,NOSYM)

Note: ATTENTION is supported only under TSO.

The TEST option will imply GONUMBER.

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

TUNE

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

```
              ┌─0─┐
►►──TUNE──(───┴─n─┴───)──────────────────────────────────────────────►◄
```

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

Specify the group to which a model number belongs as a subparameter. If you specify a model which does not exist or is not supported, a warning message is issued stating that the suboption is invalid and that the default will be used.

The following current models are supported:

0 Generates code that is executable on all models, but it will not be able to take advantage of architectural differences on the models specified below.

1 Generates code that is executable on all models but is optimized for the following models:

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

9021-xx1 and 9021-xx2

2 Generates code that is executable on all models but is optimized for the following and follow-on models:

9672-Rx2, 9672-Rx3, 9672-Rx4, and 2003

9672-Rx1, 9672-Exx, and 9672-Pxx

3 Produces code that is optimized for the 9672 G5 models

WIDECHAR

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

```
                     ┌─BIGENDIAN────┐
►►──WIDECHAR──(──────┴─LITTLEENDIAN─┴───)────────────────────────────►◄
```

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.

WINDOW

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

```
           ┌─1950─┐
►►──WINDOW──(──┴─w────┴──)──────────────────────────────────────►◄
```

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

XREF

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

```
    ┌─NOXREF──────────────────────────┐
►►──┴───XREF──┬──────────────────────┬┴─────────────────────────►◄
              │    ┌─FULL──┐         │
              └─(──┴─SHORT─┴──)──────┘
```

ABBREVIATIONS: X, NX

FULL
: Includes all identifiers and attributes in the compiler listing.

SHORT
: Omits unreferenced identifiers from the compiler listing.

The only names not included in the cross reference listing created when using the XREF option are label references on END statements. For example, assume that statement number 20 in the procedure PROC1 is END PROC1;. In this situation, statement number 20 does not appear in the cross reference listing for PROC1.)

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

For a description of the format and content of the cross-reference table, see "Cross-reference table" on page 43.

For more information about sorting identifiers and storage requirements with DBCSOS, see "ATTRIBUTE and cross-reference table" on page 43.

Specifying options in the %PROCESS or *PROCESS statements

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

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

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

```
%PROCESS options;
```

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

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

```
%PROCESS;
```

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

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

Compile-time options, their abbreviated syntax, and their IBM-supplied defaults are shown in Table 3 on page 4.

Using the preprocessor

The preprocessing facilities of the compiler are described in *PL/I Language Reference*. You can include statements in your PL/I program that, when executed by the preprocessor stage of the compiler, modify the source program or cause additional source statements to be included from a library. The following discussion provides some illustrations on the use of the preprocessor and explains how to establish and use source statement libraries.

Invoking the preprocessor

The compile-time option MACRO invokes the preprocessor stage of the compiler.

Table 5. Format of the preprocessor output

Column 1	Printer control character, if any, transferred from the position specified in the MARGINS option.
Columns 2-72	Source program. If the original source program used more than 71 columns, additional lines are included for any lines that need continuation. If the original source program used fewer than 71 columns, extra blanks are added on the right.

Three other compile-time options, MDECK, INSOURCE, and SYNTAX, are meaningful only when you also specify the MACRO option. For more information about these options, see MDECK on page 22, INSOURCE on page 17, and SYNTAX on page 32.

Macro facility options

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

```
►►──PP──(──MACRO──(──'──┬─────────────────────────────────┬──┬─────────────────────────────┬──►
                        │                ┌─DECIMAL─┐       │  │               ┌─UPPER─┐     │
                        └─FIXED──(───────┴─BINARY──┴───)───┘  └─CASE──(───────┴─ASIS──┴───)─┘

►──'──)──)─────────────────────────────────────────────────────────────────────────────────►◄
```

FIXED (DECIMAL or BINARY)
This option specifies the default base for FIXED variables as either DECIMAL or BINARY. (See Language Reference for more information).

CASE (ASIS or UPPER)
This option specifies if the input text is converted to uppercase. ASIS specifies that the input text is left "as is". UPPER specifies that the input text is converted to upper case.

A simple example of the use of the preprocessor to produce a source deck is shown in Figure 1 on page 39. According to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine (CITYSUB) or a function (CITYFUN). The DSNAME used for SYSPUNCH specifies a source program library on which the preprocessor output will be placed. Normally compilation would continue and the preprocessor output would be compiled.

```
//OPT4#8  JOB
//STEP2 EXEC  IBMZC,PARM.PLI='MACRO,MDECK,NOCOMPILE,NOSYNTAX'
//PLI.SYSPUNCH DD  DSNAME=HPU8.NEWLIB(FUN),DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN  DD *
 /* GIVEN ZIP CODE, FINDS CITY                                          */
 %DCL USE CHAR;
 %USE = 'FUN'                 /* FOR SUBROUTINE, %USE = 'SUB' */ ;
 %IF USE = 'FUN' %THEN %DO;
 CITYFUN: PROC(ZIPIN) RETURNS(CHAR(16)) REORDER; /* FUNCTION           */
                      %END;
                %ELSE %DO;
 CITYSUB: PROC(ZIPIN, CITYOUT) REORDER;     /* SUBROUTINE               */
   DCL CITYOUT CHAR(16);         /* CITY NAME                           */
                      %END;
   DCL (LBOUND, HBOUND) BUILTIN;
   DCL ZIPIN PIC '99999';       /* ZIP CODE                             */
   DCL 1 ZIP_CITY(7) STATIC,     /* ZIP CODE - CITY NAME TABLE          */
         2 ZIP PIC '99999' INIT(
                     95141, 95014, 95030,
                     95051, 95070, 95008,
                     0),           /* WILL NOT LOOK AT LAST ONE         */
         2 CITY CHAR(16) INIT(
                     'SAN JOSE', 'CUPERTINO', 'LOS GATOS',
                     'SANTA CLARA', 'SARATOGA', 'CAMPBELL',
                     'UNKNOWN CITY');  /* WILL NOT LOOK AT LAST ONE  */
   DCL I FIXED BIN(31);
   DO I = LBOUND(ZIP,1) TO            /* SEARCH FOR ZIP IN TABLE    */
          HBOUND(ZIP,1)-1             /* DON'T LOOK AT LAST ELEMENT */
          WHILE(ZIPIN ¬= ZIP(I));
   END;
 %IF USE = 'FUN' %THEN %DO;
   RETURN(CITY(I));                   /* RETURN CITY NAME             */
                      %END;
                %ELSE %DO;
   CITYOUT=CITY(I);                   /* RETURN CITY NAME             */
                      %END;
 END;
```

Figure 1. Using the preprocessor to produce a source deck that is placed on a source program library

Using the %INCLUDE statement

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

For an OS/390 environment

A *library* is an OS/390 partitioned data set that can be used to store other data sets called members. Source text that you might want to insert into a PL/I program using a %INCLUDE statement must exist as a member within a library. “Source Statement Library (SYSLIB)” on page 67 further describes the process of defining a source statement library to the compiler.

The statement:

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

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

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

For an OS/390 UNIX environment

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

The compiler searches for include files in the following order:

1. Current directory
2. Directories specified with the `-I` flag or INCDIR compile-time option
3. The /usr/include directory

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

A %PROCESS statement in source text included by a %INCLUDE statement results in an error in the compilation.

Figure 2 shows the use of a %INCLUDE statement to include the source statements for FUN in the procedure TEST. The library HPU8.NEWLIB is defined in the DD statement with the qualified name PLI.SYSLIB, which is added to the statements of the cataloged procedure for this job. Since the source statement library is defined by a DD statement with the name SYSLIB, the %INCLUDE statement need not include a ddname.

It is not necessary to invoke the preprocessor if your source program, and any text to be included, does not contain any macro statements.

```
//OPT4#9     JOB
//STEP3      EXEC IBMZCBG,PARM.PLI='INC,S,A,X,NEST'
//PLI.SYSLIB DD DSN=HPU8.NEWLIB,DISP=OLD
//PLI.SYSIN  DD *
   TEST: PROC OPTIONS(MAIN) REORDER;
     DCL ZIP PIC '99999';          /* ZIP CODE                    */
     DCL EOF BIT INIT('0'B);
     ON ENDFILE(SYSIN) EOF = '1'B;
     GET EDIT(ZIP) (COL(1), P'99999');
     DO WHILE(¬EOF);
       PUT SKIP EDIT(ZIP, CITYFUN(ZIP)) (P'99999', A(16));
       GET EDIT(ZIP) (COL(1), P'99999');
     END;
     %PAGE;
     %INCLUDE FUN;
   END;                            /* TEST                        */
//GO.SYSIN DD *
95141
95030
94101
//
```

Figure 2. Including source statements from a library

Using % statements

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

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

%INCLUDE	Directs the compiler to incorporate external strings of characters and/or graphics into the source program.
%PRINT	Directs the compiler to resume printing the source and insource listings.
%NOPRINT	Directs the compiler to suspend printing the source and insource listings until a %PRINT statement is encountered.
%OPTION	Specifies one of a selected subset of compiler options for a segment of source code.
%PAGE	Directs the compiler to print the statement immediately after a %PAGE statement in the program listing on the first line of the next page.
%POP	Directs the compiler to restore the status of the %PRINT, %NOPRINT, and %OPTION saved by the most recent %PUSH.
%PUSH	Saves the current status of the %PRINT, %NOPRINT, and %OPTION in a *push down* stack on a last-in, first-out basis.
%SKIP	Specifies the number of lines to be skipped.

Using the compiler listing

During compilation, the compiler generates a listing, most of which is optional, that contains information about the source program, the compilation, and the object module. The following description of the listing refers to its appearance on a printed page.

Of course, if compilation terminates before reaching a particular stage of processing, the corresponding listings do not appear.

Heading information

The first page of the listing is identified by the product number, the compiler version number, and the date and the time compilation commenced. This page and subsequent pages are numbered.

Near the end of the listing you will find either a statement that no errors or warning conditions were detected during the compilation, or a message that one or more errors were detected. The format of the messages is described under "Messages and return codes" on page 46. The second to the last line of the listing shows the CPU time taken for the compilation. The last line of the listing is *END OF COMPILATION OF xxxx*. where *xxxx* is the external procedure name. If you specify the NOSYNTAX compile-time option, or the compiler aborts early in the compilation, the external procedure name *xxxx* is not included and the line truncates to *END OF COMPILATION*.

The following sections describe the optional parts of the listing in the order in which they appear.

Options used for compilation

If you specify the OPTIONS option, a complete list of the options specified for the compilation, including the default options, appears on the first pages.

Preprocessor input

If you specify both the MACRO and INSOURCE options, the compiler lists input to the preprocessor, one record per line, each line numbered sequentially at the left.

If the preprocessor detects an error, or the possibility of an error, it prints a message on the page or pages following the input listing. The format of these messages is the same as the format for the compiler messages described under "Messages and return codes" on page 46.

SOURCE program

If you specify the SOURCE option, the compiler lists one record per line. If the input records contain printer control characters, or %SKIP or %PAGE statements, the lines are spaced accordingly. Use %NOPRINT and %PRINT statements to stop and restart the printing of the listing.

If you specify the MACRO option, the source listing shows the included text in place of the %INCLUDE statements in the primary input data set.

Statement nesting level

If you specify the NEST option, the block level and the DO-level are printed to the right of the statement or line number under the headings LEV and NT respectively, as in the following example:

```
Line.File LV NT
   1.1            A: PROC OPTIONS(MAIN);
   2.1     1      B: PROC;
   3.1     2         DCL K(10,10) FIXED BIN (15);
   4.1     2         DCL Y FIXED BIN (15) INIT (6);
   5.1     2         DO I=1 TO 10;
   6.1     2  1         DO J=1 TO 10;
   7.1     2  2            K(I,J) = N;
   8.1     2  2         END;
   9.1     2  1      BEGIN;
  10.1     3  1         K(1,1)=Y;
  11.1     3  1      END;
  12.1     2  1   END B;
  13.1     1      END A;
```

ATTRIBUTE and cross-reference table

If you specify the ATTRIBUTES option, the compiler prints an attribute table containing a list of the identifiers in the source program together with their declared and default attributes.

If you specify the XREF option, the compiler prints a cross-reference table containing a list of the identifiers in the source program together with the file and line numbers of the statements in which they appear.

If you specify both ATTRIBUTES and XREF, the two tables are combined. In these tables, if you explicitly declare an identifier, the compiler will list file number and line number of its DECLARE. Contextually declared variables are marked by +++++, and other implicitly declared variables are marked by *****.

Attribute table

The compiler never includes the attributes INTERNAL and REAL. You can assume them unless the respective conflicting attributes, EXTERNAL and COMPLEX, appear.

For a file identifier, the attribute FILE always appears, and the attribute EXTERNAL appears if it applies; otherwise, the compiler only lists explicitly declared attributes.

The compiler prints the dimension attribute for an array first. It prints the bounds as in the array declaration, but expressions are replaced by asterisks unless they have been reduced by the compiler to a constant, in which case the value of the constant is shown.

For a character string, a bit string, a graphic string, or an area variable, the compiler prints the length, as in the declaration, but expressions are replaced by asterisks unless they have been reduced by the compiler to a constant, in which case the value of the constant is shown.

Cross-reference table

If you combine the cross-reference table with the attribute table, the list of attributes for a name is identified by file number and line number. An identifier appears in the `Sets:` part of the cross-reference table if it is:

- The target of an assignment statement
- Used as a loop control variable in DO loops
- Used in the SET option of an ALLOCATE or LOCATE statement
- Used in the REPLY option of a DISPLAY statement

If you specify ATTRIBUTES and XREF, the two tables are combined.

If there are unreferenced identifiers, they are displayed in a separate table.

Aggregate length table

An aggregate length table is obtained by using the AGGREGATE option. The table shows how the compiler maps each aggregate with constant extents in the program. It contains the following information:

- The file and line number where the aggregate is declared.
- The name of the aggregate and each element within the aggregate.

- The byte offset of each element from the beginning of the aggregate. As a word of caution, be careful when interpreting the data offsets indicated in the data length table. An odd offset does not necessarily represent a data element without halfword, fullword, or even double word alignment. If you specify or infer the aligned attribute for a structure or its elements, the proper alignment requirements are consistent with respect to other elements in the structure, even though the table does not indicate the proper alignment relative to the beginning of the table.
- The length of each element.
- The total length of each aggregate, structure, and substructure.

If there is padding between two structure elements, a /*PADDING*/ comment appears, with appropriate diagnostic information.

Statement offset addresses

If the LIST compile option is used, the compiler includes a pseudo-assembler listing in the compiler listing. The offset given in the run-time error messages can be used to determine the erroneous statement. The offset in the run-time message is an entry offset, and can be easily found in the compiler listing when the OFFSET compile option is used; without the OFFSET compile option, the offsets given in the pseudo-assembler listing will be relative to the compile-unit entry address, and some simple calculations are required to match the run-time message entry offset to the offsets in the listing. It is for this reason we recommend the use of the OFFSET compile option.

In the example shown in Figure 3 on page 45, the message indicates that the condition was raised at offset +58 from the SUB1 entry. The compiler listing excerpt shows this offset associated with line number 8. This listing was produced using the OFFSET compile option. The runtime output from this erroneous statement is shoiwn if Figure 4 on page 45.

```
Compiler Source
  Line.File
     2.1      TheMain: proc options( main );
     3.1        call sub1();
     4.1        Sub1: proc;
     5.1          dcl (i, j) fixed bin(31);
     6.1
     7.1          i = 0;
     8.1          j = j / i;
     9.1        end Sub1;
    10.1      end TheMain;

 . . .

OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G
000000                    00002 |        THEMAIN  DS    0D

 . . .

00004C  5800  C1F4        00002 |                 L     r0,_CEECAA_(,r12,500)
000050  5000  D098        00002 |                 ST    r0,#_CEECAACRENT_1(,r13,152)
000054  5810  D098        00000 |                 L     r1,#_CEECAACRENT_1(,r13,152)
000058  5820  3062        00000 |                 L     r2,=Q(@STATIC)(,r3,98)
00005C  4152  1000        00000 |                 LA    r5,=Q(@STATIC)(r2,r1,0)
000060  18BD              00003 |                 LR    r11,r13
000062  5800  D098        00003 |                 L     r0,#_CEECAACRENT_1(,r13,152)
000066  5000  C1F4        00003 |                 ST    r0,_CEECAA_(,r12,500)
00006A  58F0  3066        00003 |                 L     r15,=A(SUB1)(,r3,102)
00006E  05EF              00003 |                 BALR  r14,r15
000070                    00010 |        @1L1     DS    0H
000070  5810  5000        00010 |                 L     r1,IBMQEFSH(,r5,0)
000074  58F0  1008        00010 |                 L     r15,&Func_&WSA(,r1,8)
000078  5800  100C        00010 |                 L     r0,&Func_&WSA(,r1,12)
00007C  5000  C1F4        00010 |                 ST    r0,_CEECAA_(,r12,500)
000080  05EF              00010 |                 BALR  r14,r15
000082                    00010 |        @1L4     DS    0H
000082  5800  D098        00002 |                 L     r0,#_CEECAACRENT_1(,r13,152)
000086  5000  C1F4        00002 |                 ST    r0,_CEECAA_(,r12,500)

 . . .

000000                    00004 |        SUB1     DS    0D

 . . .

000048  4100  0000        00007 |                 LA    r0,0
00004C  5000  D098        00007 |                 ST    r0,I(,r13,152)
000050  5840  D09C        00008 |                 L     r4,J(,r13,156)
000054  8E40  0020        00008 |                 SRDA  r4,32
000058  1D40              00008 |                 DR    r4,r0
00005A  1805              00008 |                 LR    r0,r5
00005C  5000  D09C        00008 |                 ST    r0,J(,r13,156)
```

Figure 3. Finding statement number (compiler listing example)

```
Message :

IBM0301S ONCODE=320  The ZERODIVIDE condition was raised.
         From entry point SUB1 at compile unit offset +00000058 at
         address 0D3012C0.
```

Figure 4. Finding statement number (runtime message example)

Entry offsets given in dump and ON-unit SNAP error messages can be compared with this table and the erroneous statement discovered. The statement is identified by finding the section of the table that relates to the block named in the message and then finding the largest offset less than or equal to the offset in the message. The statement number associated with this offset is the one needed.

Messages and return codes

If the preprocessor or the compiler detects an error, or the possibility of an error, messages are generated. Messages generated by the preprocessor appear in the listing immediately after the listing of the statements processed by the preprocessor. You can generate your own messages in the preprocessing stage by use of the %NOTE statement. Such messages might be used to show how many times a particular replacement had been made. Messages generated by the compiler appear at the end of the listing.

Messages are displayed in the following format:

```
PPPnnnnI X
```

where `PPP` is the prefix identifying the origin of the message (for example, IBM indicates the PL/I compiler), `nnnn` is the 4-digit message number, and `X` identifies the severity code. All messages are graded according to their severity, and the severity codes are I, W, E, S, and U.

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved. For OS/390, this code appears in the *end-of-step* message that follows the listing of the job control statements and job scheduler messages for each step.

Table 6 provides an explanation of the severity codes and the comparable return code for each:

Table 6. Description of PL/I error codes and return codes

Severity Code	Return Code	Message Type	Description
I	0000	Informational	The compiled program should run correctly. The compiler might inform you of a possible inefficiency in your code or some other condition of interest.
W	0004	Warning	A statement might be in error (warning) even though it is syntactically valid. The compiled program should run correctly, but it might produce different results than expected or be significantly inefficient.
E	0008	Error	A simple error fixed by the compiler. The compiled program should run correctly, but it might product different results than expected.
S	0012	Severe	An error not fixed by the compiler. If the program is compiled and an object module is produced, it should not be used.
U	0016	Unrecoverable	An error that forces termination of the compilation. An object module is not successfully created.

Note: Compiler messages are printed in groups according to these severity levels.

The compiler lists only messages that have a severity equal to or greater than that specified by the FLAG option, as shown in Table 7 on page 47.

Table 7. Using the FLAG option to select the lowest message severity listed

Type of Message	**Option**
Information	FLAG(I)
Warning	FLAG(W)
Error	FLAG(E)
Severe Error	FLAG(S)
Unrecoverable Error	Always listed

The text of each message, an explanation, and any recommended programmer response, are given in *VisualAge PL/I Compile-Time Messages and Codes*.

Chapter 2. Using PL/I cataloged procedures

This chapter describes the standard cataloged procedures supplied by IBM for use with the IBM VisualAge PL/I for OS/390 compiler. It explains how to invoke them, and how to temporarily or permanently modify them. The Language Environment SCEERUN data set must be located in STEPLIB and accessable to the compiler when you use any of the cataloged procedures described in this chapter.

A cataloged procedure is a set of job control statements, stored in a library, that includes one or more EXEC statements, each of which can be followed by one or more DD statements. You can retrieve the statements by naming the cataloged procedure in the PROC parameter of an EXEC statement in the input stream.

You can use cataloged procedures to save time and reduce Job Control Language (JCL) errors. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job. You should review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for your own conventions.

IBM-supplied cataloged procedures

The PL/I cataloged procedures supplied for use with VisualAge PL/I for OS/390 are:

IBMZC	Compile only
IBMZCB	Compile and bind
IBMZCPL	Compile, prelink, and link-edit
IBMZCBG	Compile, bind, and run
IBMZCPLG	Compile, prelink, link-edit, and run
IBMZCPG	Compile, prelink, load, and run

Cataloged procedures IBMZCB and IBMZCBG use features of the program management binder introduced in DFSMS/MVS® 1.4 in place of the prelinker supplied with Language Environment. These procedures produce a program object in a PDSE.

Cataloged procedures IBMZCPL, IBMZCPLG and IBMZCPG use the prelinker supplied with Language Environment and produce a load module in PDS. Use these procedures if you do not want to use a PDSE. The information in this section describes the procedure steps of the different cataloged procedures. For a description of the individual statements for compiling and link editing, see "Invoking the compiler under OS/390 using JCL" on page 65 and *OS/390 Language Environment Programming Guide*. These cataloged procedures do not include a DD statement for the input data set; you must always provide one. The example shown in Figure 5 on page 49 illustrates the JCL statements you might use to invoke the cataloged procedure IBMZCBG to compile, bind, and run a PL/I program.

VisualAge PL/I requires a minimum REGION size of 512K. Large programs require more storage. If you do not specify REGION on the EXEC statement that invokes the cataloged procedure you are running, the compiler uses the default REGION size for your site. The default size might or might not be adequate, depending on the size of your PL/I program. For an example of specifying REGION on the EXEC statement, see Figure 5 on page 49.

```
//COLEGO    JOB
//STEP1     EXEC IBMZCBG, REGION.PLI=1M
//PLI.SYSIN DD *
              .
              .
              .
    (insert PL/I program to be compiled here)
              .
              .
              .
/*
```

Figure 5. Invoking a cataloged procedure

Compile only (IBMZC)

The IBMZC cataloged procedure, shown in Figure 6 on page 50, includes only one procedure step, in which the options specified for the compilation are OBJECT and OPTIONS. (IBMZPLI is the symbolic name of the compiler.) In common with the other cataloged procedures that include a compilation procedure step, IBMZC does not include a DD statement for the input data set; you must always supply an appropriate statement with the qualified ddname PLI.SYSIN.

The OBJECT compile-time option causes the compiler to place the object module, in a syntax suitable for input to the linkage editor, in the standard data set defined by the DD statement with the name SYSLIN. This statement defines a temporary data set named &&LOADSET on a sequential device; if you want to retain the object module after the end of your job, you must substitute a permanent name for &&LOADSET (that is, a name that does not start with &&) and specify KEEP in the appropriate DISP parameter for the last procedure step that used the data set. You can do this by providing your own SYSLIN DD statement, as shown below. The data set name and disposition parameters on this statement will override those on the IBMZC procedure SYSLIN DD statement. In this example, the compile step is the only step in the job.

```
//PLICOMP EXEC IBMZC
//PLI.SYSLIN  DD  DSN=MYPROG,DISP=(MOD,KEEP)
//PLI.SYSIN   DD ...
```

The term MOD in the DISP parameter in Figure 6 on page 50 allows the compiler to place more than one object module in the data set, and PASS ensures that the data set is available to a later procedure step providing a corresponding DD statement is included there.

The SYSLIN SPACE parameter allows an initial allocation of 1 cylinder and, if necessary, 15 further allocations of 1 cylinder (a total of 16 cylinders).

```
//IBMZC   PROC LNGPRFX='IBMZ.V2R2M0',LIBPRFX='CEE',
//             SYSLBLK=3200
//*
//********************************************************************
//*                                                                  *
//* LICENSED MATERIALS - PROPERTY OF IBM                             *
//*                                                                  *
//* 5655-B22 (C) COPYRIGHT IBM CORP. 1999                            *
//* ALL RIGHTS RESERVED.                                             *
//*                                                                  *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                     *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                      *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                             *
//*                                                                  *
//*                                                                  *
//********************************************************************
//*
//* IBM VisualAge PL/I for OS/390 Version 2 Release 2 Modification 0
//*
//*  COMPILE A PL/I PROGRAM
//*
//*  RELEASE LEVEL: 02.02.00  (VERSION.RELEASE.MODIFICATION LEVEL)
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*
//*********************************************************************
//* COMPILE STEP
//*********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS',REGION=512K
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 6. Cataloged Procedure IBMZC

Compile and bind (IBMZCB)

The IBMZCB cataloged procedure, shown in Figure 7 on page 51, includes two procedure steps: PLI, which is identical to cataloged procedure IBMZC, and BIND, which invokes the Program Management binder (symbolic name IEWBLINK) to bind the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement BIND specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 8 (that is, if a severe or unrecoverable error occurs during compilation).

```
//IBMZCB  PROC LNGPRFX='IBMZ.V2R2M0',LIBPRFX='CEE',
//             SYSLBLK=3200,GOPGM=GO
//*
//*******************************************************************
//*                                                                 *
//* LICENSED MATERIALS - PROPERTY OF IBM                            *
//*                                                                 *
//* 5655-B22 (C) COPYRIGHT IBM CORP. 1999                           *
//* ALL RIGHTS RESERVED.                                            *
//*                                                                 *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                    *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                     *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                            *
//*                                                                 *
//*******************************************************************
//*
//* IBM VISUALAGE PL/I FOR OS/390 VERSION 2 RELEASE 2 MODIFICATION 0
//*
//*  COMPILE AND BIND A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//********************************************************************
//* COMPILE STEP
//********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS',REGION=512K
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//********************************************************************
//* BIND STEP
//********************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//             PARM='XREF,COMPAT=PM3',REGION=2048K
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
```

Figure 7. Cataloged Procedure IBMZCB

The Program Management binder always places the program objects it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the program object will be placed and given the member name GO. In specifying a temporary library, the cataloged procedure assumes that you will run the program object in the same job; if you want to retain the program object, you must substitute your own statement for the DD statement with the name SYSLMOD.

Compile, bind, and run (IBMZCBG)

The IBMZCBG cataloged procedure, shown in Figure 8, includes three procedure steps: PLI, BIND, and GO. PLI and BIND are identical to the two procedure steps of IBMZCB, and GO runs the program object created in the step BIND. The GO step is executed only if no severe or unrecoverable errors occurred in the preceding procedure steps.

Input data for the compilation procedure step should be specified in a DD statement with the name PLI.SYSIN, and for the GO step in a DD statement with the name GO.SYSIN.

```
//IBMZCBG  PROC LNGPRFX='IBMZ.V2R2M0',LIBPRFX='CEE',
//            SYSLBLK=3200,GOPGM=GO
//*
//********************************************************************
//*                                                                  *
//* LICENSED MATERIALS - PROPERTY OF IBM                             *
//*                                                                  *
//* 5655-B22 (C) COPYRIGHT IBM CORP. 1999                            *
//* ALL RIGHTS RESERVED.                                             *
//*                                                                  *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                     *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                      *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                             *
//*                                                                  *
//********************************************************************
//*
//* IBM VISUALAGE PL/I FOR OS/390 VERSION 2 RELEASE 2 MODIFICATION 0
//*
//*  COMPILE, BIND, AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   GOPGM     GO               MEMBER NAME FOR PROGRAM OBJECT
//*
//*********************************************************************
//* COMPILE STEP
//*********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS',REGION=512K
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 8 (Part 1 of 2). Cataloged Procedure IBMZCBG

```
//********************************************************************
//* BIND STEP
//********************************************************************
//BIND     EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//             PARM='XREF,COMPAT=PM3',REGION=2048K
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD DD  DUMMY
//SYSIN    DD  DUMMY
//********************************************************************
//* RUN STEP
//********************************************************************
//GO       EXEC PGM=*.BIND.SYSLMOD,COND=((8,LT,PLI),(8,LE,BIND)),
//             REGION=2048K
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 8 (Part 2 of 2). Cataloged Procedure IBMZCBG

Compile, prelink, and link-edit (IBMZCPL)

The IBMZCPL cataloged procedure, shown in Figure 9, includes three procedure steps: PLI, which is identical to cataloged procedure IBMZC; PLKED, which invokes the Language Environment prelinker; and LKED, which invokes the linkage editor (symbolic name IEWL) to link-edit the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement LKED specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 8 (that is, if a severe or unrecoverable error occurs during compilation).

```
//IBMZCPL PROC LNGPRFX='IBMZ.V2R2M0',LIBPRFX='CEE',
//             SYSLBLK=3200,PLANG=EDCPMSGE,GOPGM=GO
//*
//*******************************************************************
//*                                                                 *
//* LICENSED MATERIALS - PROPERTY OF IBM                            *
//*                                                                 *
//* 5655-B22 (C) COPYRIGHT IBM CORP. 1999                           *
//* ALL RIGHTS RESERVED.                                            *
//*                                                                 *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                    *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                     *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                            *
//*                                                                 *
//*******************************************************************
```

```
//*
//* IBM VISUALAGE PL/I FOR OS/390 VERSION 2 RELEASE 2 MODIFICATION 0
//*
//*  COMPILE, PRELINK, LINK-EDIT A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MEMBER NAME
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//********************************************************************
//* COMPILE STEP
//********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS',REGION=512K
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//********************************************************************
//* PRE-LINK-EDIT STEP
//********************************************************************
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI),REGION=2048K
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSN=&&PLNK,DISP=(,PASS),
//             UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//********************************************************************
//* LINK-EDIT STEP
//********************************************************************
//LKED     EXEC PGM=IEWL,PARM='XREF',COND=((8,LT,PLI),(8,LE,PLKED)),
//             REGION=2048K
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1))
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,SPACE=(1024,(200,20)),
//             DCB=BLKSIZE=1024
//SYSIN    DD  DUMMY
```

Figure 9 (Part 2 of 2). Cataloged Procedure IBMZCPL

The linkage editor always places the load modules it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the load module will be placed and given the member name GO. In specifying a temporary library, the cataloged procedure assumes that you will run the load module in the same job; if you want to retain the module, you must substitute your own statement for the DD statement with the name SYSLMOD.

The SYSLIN DD statement in Figure 9 on page 53 shows how to concatenate a data set defined by a DD statement named SYSIN with the primary input (SYSLIN) to the linkage editor. You could place linkage editor control statements in the input stream by this means, as described in the *OS/390 Language Environment Programming Guide*.

Compile, prelink, link-edit, and run (IBMZCPLG)

The IBMZCPLG cataloged procedure, shown in Figure 10, includes four procedure steps: PLI, PLKED, LKED, and GO. PLI, PLKED, and LKED are identical to the three procedure steps of IBMZCPL, and GO runs the load module created in the step LKED. The GO step is executed only if no severe or unrecoverable errors occurred in the preceding procedure steps.

Input data for the compilation procedure step should be specified in a DD statement with the name PLI.SYSIN, and for the GO step in a DD statement with the name GO.SYSIN.

```
//IBMZCPLG PROC LNGPRFX='IBMZ.V2R2M0',LIBPRFX='CEE',
//             SYSLBLK=3200,PLANG=EDCPMSGE,GOPGM=GO
//*
//********************************************************************
//*                                                                  *
//* LICENSED MATERIALS - PROPERTY OF IBM                             *
//*                                                                  *
//* 5655-B22 (C) COPYRIGHT IBM CORP. 1999                            *
//* ALL RIGHTS RESERVED.                                             *
//*                                                                  *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                     *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                      *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                             *
//*                                                                  *
//********************************************************************
//*
//* IBM VISUALAGE PL/I FOR OS/390 VERSION 2 RELEASE 2 MODIFICATION 0
//*
//*  COMPILE, PRELINK, LINK-EDIT AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MEMBER NAME
//*   GOPGM     GO               MEMBER NAME FOR LOAD MODULE
//*
//*********************************************************************
//* COMPILE STEP
//*********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS',REGION=512K
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
```

Figure 10 (Part 1 of 2). Cataloged Procedure IBMZCPLG

```
//*********************************************************************
//* PRE-LINK-EDIT STEP
//*********************************************************************
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI),REGION=2048K
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSN=&&PLNK,DISP=(,PASS),UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//*********************************************************************
//* LINK-EDIT STEP
//*********************************************************************
//LKED     EXEC PGM=IEWL,PARM='XREF',COND=((8,LT,PLI),(8,LE,PLKED)),
//             REGION=2048K
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(1024,(50,20,1))
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,SPACE=(1024,(200,20)),
//             DCB=BLKSIZE=1024
//SYSIN    DD  DUMMY
//*********************************************************************
//* RUN STEP
//*********************************************************************
//GO       EXEC PGM=*.LKED.SYSLMOD,
//             COND=((8,LT,PLI),(8,LE,PLKED),(8,LE,LKED)),
//             REGION=2048K
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 10 (Part 2 of 2). Cataloged Procedure IBMZCPLG

Compile, prelink, load and run (IBMZCPG)

The IBMZCPG cataloged procedure, shown in Figure 11 on page 57, achieves the same results as IBMZCPLG but uses the loader instead of the linkage editor. Instead of using four procedure steps (compile, prelink, link-edit, and run), it has only three (compile, prelink, and load-and-run). The third procedure step runs the loader program. The loader program processes the object module produced by the compiler and runs the resultant executable program immediately. You must provide input data for the compilation step by supplying a qualified ddname PLI.SYSIN.

The use of the loader imposes certain restrictions on your PL/I program; before using this cataloged procedure, see *OS/390 Language Environment Programming Guide*, which explains how to use the loader.

```
//IBMZCPG PROC LNGPRFX='IBMZ.V2R2M0',LIBPRFX='CEE',
//             SYSLBLK=3200,PLANG=EDCPMSGE
//*
//*******************************************************************
//*                                                                 *
//* LICENSED MATERIALS - PROPERTY OF IBM                            *
//*                                                                 *
//* 5655-B22 (C) COPYRIGHT IBM CORP. 1999                           *
//* ALL RIGHTS RESERVED.                                            *
//*                                                                 *
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,                    *
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA                     *
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                            *
//*                                                                 *
//*******************************************************************
//*
//* IBM VISUALAGE PL/I FOR OS/390 VERSION 2 RELEASE 2 MODIFICATION 0
//*
//*  COMPILE, PRELINK, LOAD AND RUN A PL/I PROGRAM
//*
//*  PARAMETER  DEFAULT VALUE    USAGE
//*   LNGPRFX   IBMZ.V2R2M0      PREFIX FOR LANGUAGE DATA SET NAMES
//*   LIBPRFX   CEE              PREFIX FOR LIBRARY DATA SET NAMES
//*   SYSLBLK   3200             BLKSIZE FOR OBJECT DATA SET
//*   PLANG     EDCPMSGE         PRELINKER MESSAGES MEMBER NAME
//*
//********************************************************************
//* COMPILE STEP
//********************************************************************
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS',REGION=512K
//STEPLIB  DD  DSN=&LNGPRFX..SIBMZCMP,DISP=SHR
//         DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//             SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//********************************************************************
//* PRE-LINK-EDIT STEP
//********************************************************************
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI),REGION=2048K
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSMSGS  DD  DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD  DUMMY
//SYSMOD   DD  DSN=&&PLNK,DISP=(,PASS),
//             UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN    DD  DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
//********************************************************************
//* LOAD AND RUN STEP
//********************************************************************
//GO       EXEC PGM=LOADER,PARM='MAP,PRINT',
//             COND=((8,LT,PLI),(8,LE,PLKED)),
//             REGION=2048K
//STEPLIB  DD  DSN=&LIBPRFX..SCEERUN,DISP=SHR
//SYSLIB   DD  DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLOUT  DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 11. Cataloged Procedure IBMZCPG

For more information on other cataloged procedures, see *OS/390 Language Environment Programming Guide*.

Invoking a cataloged procedure

To invoke a cataloged procedure, specify its name in the PROC parameter of an EXEC statement. For example, to use the cataloged procedure IBMZC, you could include the following statement in the appropriate position among your other job control statements in the input stream:

```
//stepname EXEC PROC=IBMZC
```

You do not need to code the keyword PROC. If the first operand in the EXEC statement does not begin PGM= or PROC=, the job scheduler interprets it as the name of a cataloged procedure. The following statement is equivalent to that given above:

```
//stepname EXEC IBMZC
```

If you include the parameter MSGLEVEL=1 in your JOB statement, the operating system will include the original EXEC statement in its listing, and will add the statements from the cataloged procedure. In the listing, cataloged procedure statements are identified by XX or X/ as the first two characters; X/ signifies a statement that was modified for the current invocation of the cataloged procedure.

You might be required to modify the statements of a cataloged procedure for the duration of the job step in which it is invoked, either by adding DD statements or by overriding one or more parameters in the EXEC or DD statements. For example, cataloged procedures that invoke the compiler require the addition of a DD statement with the name SYSIN to define the data set containing the source statements. Also, whenever you use more than one standard link-edit procedure step in a job, you must modify all but the first cataloged procedure that you invoke if you want to run more than one of the load modules.

Specifying multiple invocations

You can invoke different cataloged procedures, or invoke the same cataloged procedure several times, in the same job. No special problems are likely to arise unless more than one of these cataloged procedures involves a link-edit procedure step, in which case you must take the following precautions to ensure that all your load modules can be run.

When the linkage editor creates a load module, it places the load module in the standard data set defined by the DD statement with the name SYSLMOD. When the binder creates a program object, it places the program object in the PDSE defined by the DD statement with the name SYSLMOD. In the absence of a linkage editor NAME statement, the linkage editor or the binder uses the member name specified in the DSNAME parameter as the name of the module. In the standard cataloged procedures, the DD statement with the name SYSLMOD always specifies a temporary library &&GOSET with the member name GO.

If you use the cataloged procedure IBMZCBG twice within the same job to compile, bind, and run two PL/I programs, and do not name each of the two program objects that the binder creates, the first program object runs twice, and the second one not at all.

To prevent this, use one of the following methods:

- Delete the library &&GOSET at the end of the GO step. In the first invocation of the cataloged procedure at the end of the GO step, add a DD statement with the syntax:

  ```
  //GO.SYSLMOD DD DSN=&&GOSET,
  //   DISP=(OLD,DELETE)
  ```

- Modify the DD statement with the name SYSLMOD in the second and subsequent invocations of the cataloged procedure so as to vary the names of the load modules. For example:

  ```
  //BIND.SYSLMOD DD DSN=&&GOSET(GO1)
  ```

  and so on.

- Use the NAME linkage editor option to give a different name to each program object and change each job step EXEC statement to specify the running of the program object with the name for that job step.

To assign a membername to the program object, you can use the linkage editor NAME option with the DSNAME parameter on the SYSLMOD DD statement. When you use this procedure, the membername **must** be identical to the name on the NAME option if the EXEC statement that runs the program refers to the SYSLMOD DD statement for the name of the module to be run.

Another option is to give each program a different name by using GOPGM on the EXEC procedure statement. For example:

```
//  EXEC IBMZCBG,GOPGM=GO2
```

Modifying the PL/I cataloged procedures

You can modify a cataloged procedure temporarily by including parameters in the EXEC statement that invokes the cataloged procedure, or by placing additional DD statements after the EXEC statement. Temporary modifications apply only for the duration of the job step in which the procedure is invoked. They do not affect the master copy of the cataloged procedure in the procedure library.

Temporary modifications can apply to EXEC or DD statements in a cataloged procedure. To change a parameter of an EXEC statement, you must include a corresponding parameter in the EXEC statement that invokes the cataloged procedure. To change one or more parameters of a DD statement, you must include a corresponding DD statement after the EXEC statement that invokes the cataloged procedure. Although you cannot add a new EXEC statement to a cataloged procedure, you can always include additional DD statements.

EXEC statement

If a parameter of an EXEC statement that invokes a cataloged procedure has an unqualified name, the parameter applies to all the EXEC statements in the cataloged procedure. The effect on the cataloged procedure depends on the parameters, as follows:

- PARM applies to the first procedure step and nullifies any other PARM parameters.
- COND and ACCT apply to all the procedure steps.

- TIME and REGION apply to all the procedure steps and override existing values.

For example, the statement:

```
//stepname EXEC IBMZCBG,PARM='OFFSET',REGION=512K
```

- Invokes the cataloged procedure IBMZCBG.
- Substitutes the option OFFSET for OBJECT and OPTIONS in the EXEC statement for procedure step PLI.
- Nullifies the PARM parameter in the EXEC statement for procedure step BIND.
- Specifies a region size of 512K for all three procedure steps.

To change the value of a parameter in only one EXEC statement of a cataloged procedure, or to add a new parameter to one EXEC statement, you must identify the EXEC statement by qualifying the name of the parameter with the name of the procedure step. For example, to alter the region size for procedure step PLI only in the preceding example, code:

```
//stepname EXEC PROC=IBMZCBG,PARM='OFFSET',REGION.PLI=512K
```

A new parameter specified in the invoking EXEC statement overrides completely the corresponding parameter in the procedure EXEC statement.

You can nullify all the options specified by a parameter by coding the keyword and equal sign without a value. For example, to suppress the bulk of the linkage editor listing when invoking the cataloged procedure IBMZCBG, code:

```
//stepname EXEC IBMZCBG,PARM.BIND=
```

DD statement

To add a DD statement to a cataloged procedure, or to modify one or more parameters of an existing DD statement, you must include a DD statement with the form `procstepname.ddname` in the appropriate position in the input stream. If `ddname` is the name of a DD statement already present in the procedure step identified by `procstepname`, the parameters in the new DD statement override the corresponding parameters in the existing DD statement; otherwise, the new DD statement is added to the procedure step. For example, the statement:

```
//PLI.SYSIN DD *
```

adds a DD statement to the procedure step PLI of cataloged procedure IBMZC and the effect of the statement:

```
//PLI.SYSPRINT DD SYSOUT=C
```

is to modify the existing DD statement SYSPRINT (causing the compiler listing to be transmitted to the system output device of class C).

Overriding DD statements must appear after the procedure invocation and in the same order as they appear in the cataloged procedure. Additional DD statements can appear after the overriding DD statements are specified for that step.

To override a parameter of a DD statement, code either a revised form of the parameter or a replacement parameter that performs a similar function (for example, SPLIT for SPACE). To nullify a parameter, code the keyword and equal sign without a value. You can override DCB subparameters by coding only those you wish to modify; that is, the DCB parameter in an overriding DD statement does not necessarily override the entire DCB parameter of the corresponding statement in the cataloged procedures.

Chapter 3. Compiling your program

This chapter describes how to invoke the compiler under OS/390 UNIX System Services (OS/390 UNIX) and the job control statements used for compiling under OS/390. The Language Environment SCEERUN data set must be accessible to the compiler when you compile your program.

Invoking the compiler under OS/390 UNIX

To compile your program under the OS/390 UNIX environment, use the **pli** command.

```
►►──pli──┬───────────────────────────────────────────────┬──────────────────►◄
         │ ┌─────────────────────┐ ┌───────────┐        │
         └─▼─command_line_option─┴─▼─input_file─┴───────┘
```

command_line_option
You can specify a `command_line_option` in the following ways:

- **-q**option
- Option flag (usually a single letter preceded by -)

If you choose to specify compile-time options on the command line, the format differs from either setting them in your source file using %PROCESS statements. See "Specifying compile-time options under OS/390 UNIX" on page 63.

input_file
The OS/390 UNIX file specification for your program files. If you omit the extension from your file specification, the compiler assumes an extension of `.pli`. If you omit the complete path, the current directory is assumed.

Input files

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

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

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

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

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

Specifying compile-time options under OS/390 UNIX

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

Refer to “Compile-time option descriptions” on page 4 for a description of these options.

When you specify options on the command line, they override the default settings of the option. They are overridden by options set in the source file.

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

- **-q***option_keyword* (compiler-specific)
- Single and multiletter flags
- -q@/u/myopts.txt

-qoption_keyword

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

```
►►──-q──option_keyword──┬──────────────────────────────────────┬──────────────►◄
                        │      ┌─:──────────────────────────┐  │
                        └─ = ──▼─┬─suboption──────────────┬─┴──┘
                                 └─suboption=argument─────┘
```

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

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

An option, for example, that contains multiple suboptions is RULES (“RULES” on page 27). To specify RULES(LAXDCL) on the command line, you would enter:

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

The LIMITS option (“LIMITS” on page 19) is slightly more complex since each of its suboptions also has an argument. You would specify LIMITS(EXTNAME(31),FIXEDDEC(15)) on the command line as shown in the following example:

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

Single and multiletter flags

The OS/390 UNIX family of compilers uses a number of common conventional flags. Each language has its own set of additional flags.

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

```
pli samp.pli -I/home/test3/include
```

In this case, `/home/test3/include` is an include directory to be searched for INCLUDE files.

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

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

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

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

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

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

```
pli -OgI/home/test3/include
```

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

```
pli -O -g -I/home/test3/include
```

Table 8. Compile-time option flags supported by VisualAge PL/I under OS/390 UNIX

Option	Description
`-c`	Compile only.
`-e`	Create names and entries for a FETCHable load module.
`-g`	Produce symbolic information used by the debugger. This option is equivalent to -qGN.
`-I`*<dir>**	Add path *<dir>* to the directories to be searched for INCLUDE files. `-I` must be followed by a path and only a single path is allowed per `-I` option. To add multiple paths, use multiple `-I` options. There shouldn't be any spaces between `-I` and the path name.
`-O`, `-O2`	Optimize generated code. This option is equivalent to -qOPT=2.
`-q`*<option>**	Pass it to the compiler. *<option>* is a compile-time option. Each option should be delimited by a comma and each suboption should be delimited by an equal sign or colon. There shouldn't be any spaces between **-q** and *<option>*.
`-v`	Display compile and link steps and execute them.
`-#`	Display compile and link steps, but do not execute them.

Note: *You must specify an argument where indicated; otherwise, the results are unpredictable.

Invoking the compiler under OS/390 using JCL

Although you will probably use cataloged procedures rather than supply all the JCL required for a job step that invokes the compiler, you should be familiar with these statements so that you can make the best use of the compiler and, if necessary, override the statements of the cataloged procedures.

The following section describes the JCL needed for compilation. The IBM-supplied cataloged procedures described in “IBM-supplied cataloged procedures” on page 48 contain these statements. You need to code them yourself only if you are not using the cataloged procedures.

EXEC statement

The basic EXEC statement is:

```
//stepname EXEC PGM
```

512K is required for the REGION parameter of this statement. The PARM parameter of the EXEC statement can be used to specify one or more of the optional facilities provided by the compiler. These facilities are described under “Specifying options in the EXEC statement” on page 68. See Chapter 1, “Using compile-time options and facilities” on page 4 for a description of the options.

DD statements for the standard data sets

The compiler requires several standard data sets, the number of data sets depends on the optional facilities specified. You must define these data sets in DD statements with the standard ddnames shown, together with other characteristics of the data sets, in Table 9 on page 66. The DD statements SYSIN, SYSUT1, and SYSPRINT are always required.

You can store any of the standard data sets on a direct-access device, but you must include the SPACE parameter in the DD statement. This parameter defines the data set to specify the amount of auxiliary storage required. The amount of auxiliary storage allocated in the IBM-supplied cataloged procedures should suffice for most applications.

Table 9. Compiler standard data sets

Standard DDNAME	Contents of data set	Possible device classes[1]	Record format (RECFM)[2]	Record size (LRECL)[3]	BLKSIZE
SYSIN	Input to the compiler	SYSSQ	F,FB,U VB,V	<101(100) <105(104)	—
SYSLIN	Object module	SYSSQ	FB	80	80
SYSPUNCH	Preprocessor output, compiler output	SYSSQ SYSCP	FB	80	80
SYSUT1	Temporary workfile	SYSDA	F	4051	—
SYSPRINT	Listing, including messages	SYSSQ	VBA	125	129
SYSLIB	Source statements for preprocessor	SYSDA	F,FB,U V,VB	<101 <105	—

Notes:

The only value for compile-time SYSPRINT that can be overridden is BLKSIZE.

1. The possible device classes are:

SYSSQ	Sequential device
SYSDA	Direct-access device
SYSCP	Card-punch device.

   Block size can be specified except for SYSUT1. The block size and logical record length for SYSUT1 is chosen by the compiler.

2. If the record format is not specified in a DD statement, the default value is provided by the compiler. (Default values are shown in italics.)

3. The numbers in parentheses in the “Record Size” column are the defaults, which you can override.

Input (SYSIN)

Input to the compiler must be a data set defined by a DD statement with the name SYSIN. This data set must have CONSECUTIVE organization. The input must be one or more external PL/I procedures. If you want to compile more than one external procedure in a single job or job step, precede each procedure, except possibly the first, with a %PROCESS statement.

80-byte records are commonly used as the input medium for PL/I source programs. The input data set can be on a direct-access device or some other sequential media. The input data set can contain either fixed-length records (blocked or unblocked), variable-length records (coded or uncoded), or undefined-length records. The maximum record size is 100 bytes.

When data sets are concatenated for input to the compiler, the concatenated data sets must have similar characteristics (for example, block size and record format).

Output (SYSLIN, SYSPUNCH)

Output in the form of one or more object modules from the compiler can be stored in the data set SYSLIN if you specify the OBJECT compile-time option. This data set is defined by the DD statement.

The object module is always in the form of 80-byte fixed-length records, blocked or unblocked. The data set defined by the DD statement with the name SYSPUNCH is also used to store the output from the preprocessor if you specify the MDECK compile-time option.

Temporary workfile (SYSUT1)

The compiler requires a data set for use as a temporary workfile. It is defined by a DD statement with the name SYSUT1, and is known as the *spill file*. It must be on a direct-access device, and must not be allocated as a multi-volume data set.

The spill file is used as a logical extension to main storage and is used by the compiler and by the preprocessor to contain text and dictionary information. The LRECL and BLKSIZE for SYSUT1 is chosen by the compiler based on the amount of storage available for spill file pages.

The DD statements given in this publication and in the cataloged procedures for SYSUT1 request a space allocation in blocks of 1024 bytes. This is to insure that adequate secondary allocations of direct-access storage space are acquired.

Listing (SYSPRINT)

The compiler generates a listing that includes all the source statements that it processed, information relating to the object module, and, when necessary, messages. Most of the information included in the listing is optional, and you can specify those parts that you require by including the appropriate compile-time options. The information that can appear, and the associated compile-time options, are described under “Using the compiler listing” on page 41.

You must define the data set, in which you wish the compiler to store its listing, in a DD statement with the name SYSPRINT. This data set must have CONSECUTIVE organization. Although the listing is usually printed, it can be stored on any sequential or direct-access device. For printed output, the following statement will suffice if your installation follows the convention that output class A refers to a printer:

```
//SYSPRINT DD SYSOUT=A
```

Source Statement Library (SYSLIB)

If you use the preprocessor %INCLUDE statement to introduce source statements into the PL/I program from a library, you can either define the library in a DD statement with the name SYSLIB, or you can choose your own ddname (or ddnames) and specify a ddname in each %INCLUDE statement. (For further information on the preprocessor, see “Using the preprocessor” on page 37.)

If the statements are included from a SYSLIB, they must have a form that is similar to the %INCLUDE statement. For example, they must have the same record format (fixed, variable, undefined), the same logical record length, and matching left and right margins.

The BLOCKSIZE of the library must be less than or equal to 32,760 bytes.

Specifying options

For each compilation, the IBM-supplied or installation default for a compile-time option applies unless it is overridden by specifying the option in a %PROCESS statement or in the PARM parameter of an EXEC statement.

An option specified in the PARM parameter overrides the default value, and an option specified in a %PROCESS statement overrides both that specified in the PARM parameter and the default value.

Note: When conflicting attributes are specified either explicitly or implicitly by the specification of other options, the latest implied or explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden in this way.

Specifying options in the EXEC statement

To specify options in the EXEC statement, code `PARM=` followed by the list of options, in any order separating the options with commas and enclosing the list within single quotation marks, for example:

```
//STEP1 EXEC PGM=IBMZPLI,PARM='OBJECT,LIST'
```

Any option that has quotation marks, for example `MARGINI('c')`, must have the quotation marks duplicated. The length of the option list must not exceed 100 characters, including the separating commas. However, many of the options have an abbreviated syntax that you can use to save space. If you need to continue the statement onto another line, you must enclose the list of options in parentheses (instead of in quotation marks) enclose the options list on each line in quotation marks, and ensure that the last comma on each line except the last line is outside of the quotation marks. An example covering all the above points is as follows:

```
//STEP1 EXEC PGM=IBMZPLI,PARM=('AG,A',
//       'C,F(I)',
// 'M,MI(''X''),NEST,STG,X')
```

If you are using a cataloged procedure, and want to specify options explicitly, you must include the PARM parameter in the EXEC statement that invokes it, qualifying the keyword PARM with the name of the procedure step that invokes the compiler. For example:

```
//STEP1 EXEC nnnnnnn,PARM.PLI='A,LIST'
```

Specifying options in the EXEC statement using options file

Another way to specify options in the EXEC statement is by declaring all your options in an options file and coding the following:

```
//STEP1 EXEC PGM==IBMZPLI,PARM='@DD:OPTIONS'
```

This method allows you to provide a consistent set of options that you frequently use. This is especially effective if you want other programmers to use a common set of options. It also gets you past the 100-character limit.

The MARGINS option does not apply to options files: the data in column 1 will be read as part of the options. Also, if the file is F-format, any data after column 72 will be ignored.

The parm string con contain "normal" options and can point to more than one options file. For instance, to specify the option LIST as well as from both the file in the GROUP DD and in the PROJECT DD, you could specify

```
  PARM='LIST @DD:GROUP @DD:PROJECT'
```

The options in the PROJECT file would have precedence over options in the GROUP file.

Also, in this case, the LIST option might be turned off by a NOLIST option specified in either of the options files. To insure that the LIST option is on, you could specify

```
PARM='@DD:GROUP @DD:PROJECT LIST'
```

Options files may also be used under USS. For example, in USS, to compile sample.pli with options from the file /u/pli/group.opt, you would specify

```
pli -q@/u/pli/group.opt  sample.pli
```

Return codes in batched compilation

The return code generated by a batched compilation is the highest code that is returned if the procedures are compiled separately.

JCL for batched processing

The only special consideration relating to JCL for batched processing refers to the data set defined by the DD statement with the name SYSLIN. If you include the option OBJECT, ensure that this DD statement contains the parameter DISP=(MOD,KEEP) or DISP=(MOD,PASS). (The IBM-supplied cataloged procedures specify DISP=(MOD,PASS).) If you do not specify DISP=MOD, successive object modules will overwrite the preceding modules.

Examples of batched compilations

If the external procedures are components of a large program and need to be run together, you can link-edit them together and run them in subsequent job steps. Cataloged procedure IBMZCG can be used, as shown in Figure 12.

```
//OPT4#13 JOB
//STEP1 EXEC IBMZCG
//PLI.SYSIN DD *
          First PL/I source program
% PROCESS;
          Second PL/I source program
% PROCESS;
          Third PL/I source program
/*
//GO.SYSIN DD *
          Data processed by combined
          PL/I programs

/*
```

Figure 12. Example of batched compilation, including execution

If the external procedures are independent programs to be invoked individually from a load module library, cataloged procedure IBMZCL can be used. For example, a job that contains three compile and link-edit operations can be run as a single batched compilation, as shown in Figure 13.

```
//OPT4#14 JOB
//STEP1 EXEC IBMZCL,
// PARM.PLI='SOURCE',
// PARM.LKED=LIST
//PLI.SYSIN DD *
          PL/I source program
/*
//LKED.SYSLMOD DD DSN=PUBPGM,
// DISP=OLD
```

Figure 13. Example of batched compilation, excluding execution

Compiling for CICS

When coding a CICS transaction in PL/I, prior to compiling your transaction, you must invoke the CICS Command Language Translator. You can find information on the CICS Command Language Translator in *CICS/ESA® Application Programmer's Reference Manual*. After the CICS translator step ends, compile your PL/I program with the SYSTEM(CICS) option. NOEXECOPS is implied with this option. For a description of the SYSTEM compile-time option, see "SYSTEM" on page 33.

Chapter 4. Link-editing and running

After compilation, your program consists of one or more object modules that contain unresolved references to each other, as well as references to the Language Environment run-time library. These references are resolved during link-editing (statically) or during execution (dynamically). There are two ways to link-edit statically:

1. Use the prelinker prior to the traditional link step
2. Link without the prelinker, which is similar to linking with PL/I for MVS & VM except that you now must use a PDSE to hold the load module prior to execution.

After you compile your PL/I program, the next step is to link and run your program with test data to verify that it produces the results you expect. When using VisualAge PL/I we recommend you select the method of linking without the prelinker (as described in Item 2 above). If you do not use a PDSE to hold your load modules, you must prelink.

Language Environment provides the run-time environment and services you need to execute your program. For instructions on linking and running PL/I and all other Language Environment-conforming language programs, refer to *OS/390 Language Environment Programming Guide*. For information about migrating your existing PL/I programs to Language Environment, see *VisualAge PL/I for OS/390 Compiler and Run-Time Migration Guide*.

Run-time considerations

You can specify run-time options as parameters passed to the program initialization routine. You can also specify run-time options in the PLIXOPT variable. It might also prove beneficial, from a performance standpoint, if you alter your existing programs by using the PLIXOPT variable to specify your run-time options and recompiling your programs. For a description of using PLIXOPT, see *Language Environment Programming Guide*.

To simplify input/output at the terminal, various conventions have been adopted for stream files that are assigned to the terminal. Three areas are affected:

1. Formatting of PRINT files
2. The automatic prompting feature
3. Spacing and punctuation rules for input.

Note: No prompting or other facilities are provided for record I/O at the terminal, so you are strongly advised to use stream I/O for any transmission to or from a terminal.

Formatting conventions for PRINT files

When a PRINT file is assigned to the terminal, it is assumed that it will be read as it is being printed. Spacing is therefore reduced to a minimum to reduce printing time. The following rules apply to the PAGE, SKIP, and ENDPAGE keywords:

- PAGE options or format items result in three lines being skipped.

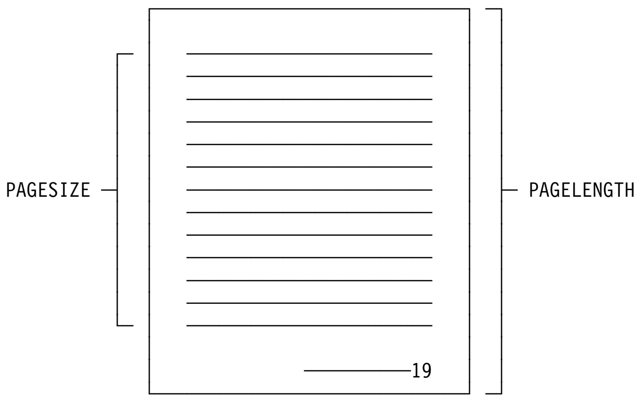
- SKIP options or format items larger than SKIP (2) result in three lines being skipped. SKIP (2) or less is treated in the usual manner.
- The ENDPAGE condition is never raised.

Changing the format on PRINT files

If you want normal spacing to apply to output from a PRINT file at the terminal, you must supply your own tab table for PL/I. This is done by declaring an external structure called PLITABS in the main program and initializing the element PAGELENGTH to the number of lines that can fit on your page. This value differs from PAGESIZE, which defines the number of lines you want to print on the page before ENDPAGE is raised (see Figure 15). If you require a PAGELENGTH of 64 lines, declare PLITABS as shown in Figure 14. For information on overriding the tab table, see “Overriding the tab control table” on page 133.

```
DCL 1 PLITABS STATIC EXTERNAL,
  ( 2   OFFSET INIT (14),
    2    PAGESIZE INIT (60),
    2    LINESIZE INIT (120),
    2    PAGELENGTH INIT (64),
    2    FILL1 INIT (0),
    2    FILL2 INIT (0),
    2    FILL3 INIT (0),
    2    NUMBER_OF_TABS INIT (5),
    2    TAB1 INIT (25),
    2    TAB2 INIT (49),
    2    TAB3 INIT (73),
    2    TAB4 INIT (97),
    2    TAB5 INIT (121)) FIXED BIN (15,0);
```

Figure 14. Declaration of PLITABS. This declaration gives the standard page size, line size and tabulating positions



```
PAGELENGTH:  the number of lines that can be printed on a page

PAGESIZE:    the number of lines that will be printed on a page
             before the ENDPAGE condition is raised
```

Figure 15. PAGELENGTH and PAGESIZE. PAGELENGTH defines the size of your paper, PAGESIZE the number of lines in the main printing area.

Automatic prompting

When the program requires input from a file that is associated with a terminal, it issues a prompt. This takes the form of printing a colon on the next line and then skipping to column 1 on the line following the colon. This gives you a full line to enter your input, as follows:

```
:
(space for entry of your data)
```

This type of prompt is referred to as a primary prompt.

Overriding automatic prompting: You can override the primary prompt by making a colon the last item in the request for the data. You cannot override the secondary prompt. For example, the two PL/I statements:

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));
```

result in the terminal displaying:

```
ENTER TIME OF PERIHELION
:        (automatic prompt)
(space for entry of data)
```

However, if the first statement has a colon at the end of the output, as follows:

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

the sequence is:

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

Note: The override remains in force for only one prompt. You will be automatically prompted for the next item unless the automatic prompt is again overridden.

Punctuating long input lines

Line continuation character: To transmit data that requires 2 or more lines of space at the terminal as one data-item, type an SBCS hyphen as the last character in each line except the last line. For example, to transmit the sentence “this data must be transmitted as one unit.” you enter:

```
:'this data must be transmitted -
+:as one unit.'
```

Transmission does not occur until you press ENTER after “unit.'”. The hyphen is removed. The item transmitted is called a “logical line.”

Note: To transmit a line whose last data character is a hyphen or a PL/I minus sign, enter two hyphens at the end of the line, followed by a null line as the next line. For example:

```
xyz--
(press ENTER only, on this line)
```

Punctuating GET LIST and GET DATA statements

For GET LIST and GET DATA statements, a comma is added to the end of each logical line transmitted from the terminal, if the programmer omits it. Thus there is no need to enter blanks or commas to delimit items if they are entered on separate logical lines. For the PL/I statement GET LIST(A,B,C); you can enter at the terminal:

```
:1
+:2
+:3
```

This rule also applies when entering character-string data. Therefore, a character string must transmit as one logical line. Otherwise, commas are placed at the break points. For example, if you enter:

```
:'COMMAS SHOULD NOT BREAK
+:UP A CLAUSE.'
```

the resulting string is: “COMMAS SHOULD NOT BREAK, UP A CLAUSE.” The comma is not added if a hyphen was used as a line continuation character.

Automatic padding for GET EDIT: For a GET EDIT statement, there is no need to enter blanks at the end of the line. The data will be padded to the specified length. Thus, for the PL/I statement:

```
GET EDIT (NAME) (A(15));
```

you can enter the 5 characters SMITH. The data will be padded with ten blanks so that the program receives the fifteen characters:

```
'SMITH          '
```

Note: A single data item must transmit as a logical line. Otherwise, the first line transmitted will be padded with the necessary blanks and taken as the complete data item.

Use of SKIP for terminal input: All uses of SKIP for input are interpreted as SKIP(1) when the file is allocated to the terminal. SKIP(1) is treated as an instruction to ignore all unused data on the currently available logical line.

ENDFILE

The end-of-file can be entered at the terminal by keying in a logical line that consists of the two characters “/*”. Any further attempts to use the file without closing it result in the ENDFILE condition being raised.

SYSPRINT considerations

The PL/I standard SYSPRINT file is shared by multiple enclaves within an application. You can issue I/O requests, for example STREAM PUT, from the same or different enclaves. These requests are handled using the standard PL/I SYSPRINT file as a file which is common to the entire application. The SYSPRINT file is implicitly closed only when the application terminates, not at the termination of the enclave.

The standard PL/I SYSPRINT file contains user-initiated output only, such as STREAM PUTs. Run-time library messages and other similar diagnostic output are directed to the Language Environment MSGFILE. See the *OS/390 V2R8 Language Environment Programming Guide* for details on redirecting SYSPRINT file output to the Language Environment MSGFILE.

To be shared by multiple enclaves within an application, the PL/I SYSPRINT file must be declared as an EXTERNAL FILE constant with a file name of SYSPRINT and also have the attributes STREAM and OUTPUT as well as the (implied) attribute of PRINT, when OPENed. This is the standard SYSPRINT file as defaulted by the compiler.

There exists only one standard PL/I SYSPRINT FILE within an application and this file is shared by all enclaves within the application. For example, the SYSPRINT file can be shared by multiple nested enclaves within an application or by a series of enclaves that are created and terminated within an application by the Language Environment preinitialization function. To be shared by an enclave within an application, the PL/I SYSPRINT file must be declared in that enclave. The standard SYSPRINT file cannot be shared by passing it as a file argument between enclaves. The declared attributes of the standard SYSPRINT file should be the same throughout the application, as with any EXTERNALly declared constant. PL/I does not enforce this rule. Both the TITLE option and the MSGFILE(SYSPRINT) option attempt to route SYSPRINT to another data set. As such, if the two options were used together, there will be a conflict and the TITLE option will be ignored.

Having a common SYSPRINT file within an application can be an advantage to applications that utilize enclaves that are closely tied together. However, since all enclaves in an application write to the same shared data set, this might require some coordination among the enclaves.

The SYSPRINT file is opened (implicitly or explicitly) when first referenced within an enclave of the application. When the SYSPRINT file is CLOSEd, the file resources are released (as though the file had never been opened) and all enclaves are updated to reflect the closed status.

If SYSPRINT is utilized in a multiple enclave application, the LINENO built-in function only returns the current line number until after the first PUT or OPEN in an enclave has been issued. This is required in order to maintain full compatibility with old programs.

The COUNT built-in function is maintained at an enclave level. It always returns a value of zero until the first PUT in the enclave is issued. If a nested child enclave is invoked from a parent enclave, the value of the COUNT built-in function is undefined when the parent enclave regains control from the child enclave.

When opened, the TITLE option can be used to associate the standard SYSPRINT file with different operating system data sets. This association is retained across enclaves for the duration of the open.

PL/I condition handling associated with the standard PL/I SYSPRINT file retains its current semantics and scope. For example, an ENDPAGE condition raised within a child enclave will only invoke an established ON-unit within that child enclave. It does not cause invocation of an ON-unit within the parent enclave.

The tabs for the standard PL/I SYSPRINT file can vary when PUTs are done from different enclaves, if the enclaves contain a user PLITABS table.

If the PL/I SYSPRINT file is utilized as a RECORD file or as a STREAM INPUT file, PL/I supports it at an individual enclave or task level, but not as a sharable file among enclaves. If the PL/I SYSPRINT file is open at the same time with different file attributes (e.g. RECORD and STREAM) in different enclaves of the same application, results are unpredictable.

Using FETCH in your routines

In VisualAge PL/I, you can FETCH VisualAge PL/I routines, OS/390 C DLLs, and assembler routines.

FETCHing VisualAge PL/I routines

Almost all the restrictions on PL/I for MVS & VM FETCHed modules have been removed, so a FETCHed module can now:

- FETCH other modules
- Perform any I/O operations on any PL/I file. The file can be opened either by the FETCHed module, by the main module, or by some other FETCHed module.
- ALLOCATE and FREE its own CONTROLLED variables

There are, however, a few restrictions on a VisualAge PL/I module that is to be FETCHed. These restrictions are:

1. The ENTRY declaration in the routine that FETCHes must not specify OPTIONS(COBOL) or OPTIONS(ASM)—these should be specified only for COBOL or ASM routines not linked as DLLs.
2. OPTIONS(FETCHABLE) must be specified on the PROCEDURE statement for the entry point of the DLL or the procedure must be compiled with the DLLINIT option.
3. PROCEDURE statements specifiying OPTIONS(FETCHABLE) must be linked as a DLL.

As an illustration of these restrictions, consider the compiler user exit. If you specify the EXIT compile-time option, the compiler will FETCH and call a VisualAge PL/I module named IBMUEXIT.

In accordance with Item 1 above, the DECLARE in the compiler for this routine looks like:

```
dcl ibmuexit ext entry( pointer byvalue, pointer byvalue );
```

In accordance with Item 2 above, the PROCEDURE statement for this routine looks like:

```
ibmuexit:
  proc ( addr_Userexit_Interface_Block,
         addr_Request_Area )
  options( fetchable  );

  dcl addr_Userexit_Interface_Block  pointer byvalue;

  dcl addr_Request_Area              pointer byvalue;
```

In accordance with Item 3 above, the linker option DYNAM=DLL must be specified when linking the user exit into a DLL. The DLL must be linked either into a PDSE or into a temporary dataset (in which case DSNTYPE=LIBRARY must be specified on the SYSLMOD DD statement).

All the JCL to compile, link, and invoke the user exit is given in the JCL below. The one significant difference between the sample below and the code excerpts above is that, in the code below, the FETCHed user exit does not receive two BYVALUE pointers to structures, but instead it receives the two structures BYADDR. In order to make this change work, the code specifies OPTIONS(NODESCRIPTOR) on each of its PROCEDURE statements.

```
//*
//********************************************************************
//* compile the user exit
//********************************************************************
//PLIEXIT EXEC PGM=IBMZPLI,
//            REGION=256K
//STEPLIB   DD DSN=IBMZ.V2R2M0.SIBMZCMP,DISP=SHR
//          DD DSN=IBMZ.V2R2M0.SCEERUN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*Process or('|') not('!');
*Process limits(extname(31));

 /********************************************************************/
 /*                                                                  */
 /*  NAME - IBMUEXIT.PLI                                             */
 /*                                                                  */
 /*  DESCRIPTION                                                     */
 /*    User-exit sample program.                                     */
 /*                                                                  */
 /*    Licensed Materials - Property of IBM                          */
 /*    5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,2000.         */
 /*    All Rights Reserved.                                          */
 /*    US Government Users Restricted Rights-- Use, duplication or   */
 /*    disclosure restricted by GSA ADP Schedule Contract with       */
 /*    IBM Corp.                                                     */
 /*                                                                  */
 /*  DISCLAIMER OF WARRANTIES                                        */
 /*    The following menclosedy code is sample code created by IBM   */
 /*    Corporation. This sample code is not part of any standard     */
 /*    IBM product and is provided to you solely for the purpose of  */
 /*    assisting you in the development of your applications.  The   */
 /*    code is provided "AS IS", without warranty of any kind.       */
 /*    IBM shall not be liable for any damages arising out of your   */
 /*    use of the sample code, even if IBM has been advised of the   */
 /*    possibility of such damages.                                  */
 /*                                                                  */
 /********************************************************************/
```

Figure 16 (Part 1 of 8). Sample JCL to compile, link, and invoke the user exit

```
/*******************************************************************/
/*                                                                 */
/*  During initialization, IBMUEXIT is called.  It reads           */
/*  information about the messages being screened from a text      */
/*  file and stores the information in a hash table.  IBMUEXIT     */
/*  also sets up the entry points for the message filter service   */
/*  and termination service.                                       */
/*                                                                 */
/*  For each message generated by the compiler, the compiler       */
/*  calls the message filter registered by IBMUEXIT.  The filter   */
/*  looks the message up in the hash table previously created.     */
/*                                                                 */
/*  The termination service is called at the end of the compile    */
/*  but does nothing.  It could be enhanced to generates reports   */
/*  or do other cleanup work.                                      */
/*                                                                 */
/*******************************************************************/

pack: package exports(*);

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

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

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

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

      2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs  */
      2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs  */

      2 Uex_UIB_Exit_Routs,                    /* exit entries setat
                                                  initialization     */
        3 ( Uex_UIB_Termination,
            Uex_UIB_Message_Filter,            /* call for each msg  */
            *, *, *, * )
          limited entry (
            *,                                 /* to Uex_UIB         */
            *                                  /* to a request area  */
           );

  /*******************************************************************/
  /*                                                                 */
  /*   Request Area for Initialization exit                          */
  /*                                                                 */
  /*******************************************************************/

  Dcl 1 Uex_ISA native based( null() ),
        2 Uex_ISA_Length fixed bin(31);
```

Figure 16 (Part 2 of 8). Sample JCL to compile, link, and invoke the user exit

```
/*******************************************************************/
/*                                                                 */
/*   Request Area for Message_Filter exit                          */
/*                                                                 */
/*******************************************************************/

Dcl 1 Uex_MFA native based( null() ),
      2 Uex_MFA_Length   fixed bin(31),
      2 Uex_MFA_Facility_Id  char(3),
      2 *                    char(1),
      2 Uex_MFA_Message_no   fixed bin(31),
      2 Uex_MFA_Severity     fixed bin(15),
      2 Uex_MFA_New_Severity fixed bin(15);  /* set by exit proc   */

/*******************************************************************/
/*                                                                 */
/*   Request Area for Terminate exit                               */
/*                                                                 */
/*******************************************************************/

Dcl 1 Uex_TSA native based( null() ),
      2 Uex_TSA_Length fixed bin(31);

/*******************************************************************/
/*                                                                 */
/*   Severity Codes                                                */
/*                                                                 */
/*******************************************************************/

dcl uex_Severity_Normal            fixed bin(15) value(0);
dcl uex_Severity_Warning           fixed bin(15) value(4);
dcl uex_Severity_Error             fixed bin(15) value(8);
dcl uex_Severity_Severe            fixed bin(15) value(12);
dcl uex_Severity_Unrecoverable     fixed bin(15) value(16);

/*******************************************************************/
/*                                                                 */
/*   Return Codes                                                  */
/*                                                                 */
/*******************************************************************/

dcl uex_Return_Normal              fixed bin(15) value(0);
dcl uex_Return_Warning             fixed bin(15) value(4);
dcl uex_Return_Error               fixed bin(15) value(8);
dcl uex_Return_Severe              fixed bin(15) value(12);
dcl uex_Return_Unrecoverable       fixed bin(15) value(16);
```

Figure 16 (Part 3 of 8). Sample JCL to compile, link, and invoke the user exit

```
/*******************************************************************/
/*                                                                 */
/*   Reason Codes                                                  */
/*                                                                 */
/*******************************************************************/

dcl uex_Reason_Output               fixed bin(15) value(0);
dcl uex_Reason_Suppress             fixed bin(15) value(1);

dcl hashsize fixed bin(15) value(97);
dcl hashtable(0:hashsize-1) ptr init((hashsize) null());

dcl 1 message_item native based,
      2 message_Info,
        3 facid  char(3),
        3 msgno  fixed bin(31),
        3 newsev fixed bin(15),
        3 reason fixed bin(31),
      2 link pointer;

ibmuexit:
  proc ( ue, ia )
  options( fetchable nodescriptor);

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 ia like uex_Isa byaddr;

  ue.uex_Uib_Message_Filter = message_Filter;
  ue.uex_Uib_Termination = exitterm;

end;

message_Filter: proc ( ue, mf  )
  options( nodescriptor);

  dcl 1 ue like uex_Uib byaddr;
  dcl 1 mf like uex_Mfa byaddr;

  dcl sysuexit     file stream input env(recsize(80));
  dcl p            pointer;
  dcl bucket       fixed bin(31);
  dcl based_Chars  char(8) based;
  dcl title_Str    char(8) var;

  ue.uex_Uib_Message_Filter = message_Filter;
  ue.uex_Uib_Termination = exitterm;
```

Figure 16 (Part 4 of 8). Sample JCL to compile, link, and invoke the user exit

```
on undefinedfile(sysuexit)
begin;
  put edit ('** User exit unable to open exit file ')
           (A) skip;
  put skip;
  signal error;
end;

if ue.uex_Uib_User_Char_Len = 0 then
  do;
    open file(sysuexit);
  end;
else
  do;
    title_Str
     = substr( ue.uex_Uib_User_Char_Str->based_Chars,
               1, ue.uex_Uib_User_Char_Len );
    open file(sysuexit) title(title_Str);
  end;


on error, endfile(sysuexit)
  goto done;

allocate message_item set(p);

/*************************************************************/
/*                                                           */
/*  Skip header lines and read first data line               */
/*                                                           */
/*************************************************************/

get file(sysuexit) list(p->message_info) skip(3);


do loop;

  /***********************************************************/
  /*                                                         */
  /*  Put message information in hash table                  */
  /*                                                         */
  /***********************************************************/

  bucket = mod(p->msgno, hashsize);
  p->link = hashtable(bucket);
  hashtable(bucket) = p;

  /***********************************************************/
  /*                                                         */
  /*  Read next data line                                    */
  /*                                                         */
  /***********************************************************/

  allocate message_item set(p);
  get file(sysuexit) skip;
  get file(sysuexit) list(p->message_info);

end;
```

Figure 16 (Part 5 of 8). Sample JCL to compile, link, and invoke the user exit

```
   /*******************************************************************/
   /*                                                                 */
   /*  Clean up                                                       */
   /*                                                                 */
   /*******************************************************************/

   done:

   free p->message_Item;
   close file(sysuexit);

 end;


 message_Filter: proc ( ue, mf  );

   dcl 1 ue like uex_Uib byaddr;
   dcl 1 mf like uex_Mfa byaddr;

   dcl p pointer;
   dcl bucket fixed bin(15);

   on error snap system;

   ue.uex_Uib_Reason_Code = uex_Reason_Output;
   ue.uex_Uib_Return_Code = 0;

   mf.uex_Mfa_New_Severity = mf.uex_Mfa_Severity;


   /*******************************************************************/
   /*                                                                 */
   /*  Calculate bucket for error message                             */
   /*                                                                 */
   /*******************************************************************/

   bucket = mod(mf.uex_Mfa_Message_No, hashsize);


   /*******************************************************************/
   /*                                                                 */
   /*  Search bucket for error message                                */
   /*                                                                 */
   /*******************************************************************/

   do p = hashtable(bucket) repeat (p->link) while(p!=null())
     until (p->msgno = mf.uex_Mfa_Message_No &
            p->facid = mf.Uex_Mfa_Facility_Id);
   end;
```

Figure 16 (Part 6 of 8). Sample JCL to compile, link, and invoke the user exit

```
     if p = null() then;
     else
       do;

         /*********************************************************/
         /*                                                       */
         /*  Filter error based on information in has table       */
         /*                                                       */
         /*********************************************************/

         ue.uex_Uib_Reason_Code = p->reason;
         if p->newsev < 0 then;
         else
           mf.uex_Mfa_New_Severity = p->newsev;
       end;
   end;


   exitterm: proc ( ue, ta  );

     dcl 1 ue like uex_Uib byaddr;
     dcl 1 ta like uex_Tsa byaddr;

     ue.uex_Uib_return_Code = 0;
     ue.uex_Uib_reason_Code = 0;

   end;

 end pack;
//*********************************************************************
//* link the user exit
//*********************************************************************
//LKEDEXIT EXEC PGM=IEWL,PARM='XREF,LIST,LET,DYNAM=DLL',
//    COND=(9,LT,PLIEXIT),REGION=5000K
//SYSLIB   DD DSN=IBMZ.V2R2M0.CEE.SCEELKED,DISP=SHR
//SYSLMOD  DD DSN=&&EXITLIB(IBMUEXIT),DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(7,1,1)),DSNTYPE=LIBRARY
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(3,1)),
//            DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=X
//SYSDEFSD DD DUMMY
//SYSLIN    DD DSN=&&LOADSET,DISP=SHR
//        DD DDNAME=SYSIN
//LKED.SYSIN     DD *
  ENTRY IBMUEXIT
```

Figure 16 (Part 7 of 8). Sample JCL to compile, link, and invoke the user exit

```
//*******************************************************************
//* compile main
//*******************************************************************
//PLI EXEC PGM=IBMZPLI,PARM='F(I),EXIT',
//             REGION=256K
//STEPLIB   DD DSN=IBMZ.V2R2M0.SIBMZCMP,DISP=SHR
//          DD DSN=IBMZ.V2R2M0.SCEERUN,DISP=SHR
//          DD  DSN=&&EXITLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=&&LOADSET2,DISP=(MOD,PASS),UNIT=SYSSQ,
//             SPACE=(CYL,(3,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//             SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN     DD *
*process;
 MainFet: Proc Options(Main);
   /* the exit will suppress the message for the next dcl */
   dcl one_byte_integer  fixed bin(7);
 End ;
//*
//SYSUEXIT DD *
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+-------------------------------
  'IBM'     1042       -1         1      String spans multiple lines
  'IBM'     1044       -1         1      FIXED BIN 7 mapped to 1 byte
```

Figure 16 (Part 8 of 8). Sample JCL to compile, link, and invoke the user exit

FETCHing OS/390 C routines

The ENTRY declaration in the routine that FETCHes an OS/390 C routine must not specify OPTIONS(COBOL) or OPTIONS(ASM)—these should be specified only for COBOL or ASM routines not linked as DLLs

The OS/390 C documentation provides instructions on how to compile and link an OS/390 C DLL.

FETCHing assembler routines

The ENTRY declaration in the routine that FETCHes an assembler routine must specify OPTIONS(ASM).

Part 3. Using I/O facilities

Chapter 5. Using data sets and files

Your PL/I programs process and transmit units of information called *records*. A collection of records is called a *data set*. Data sets are physical collections of information external to PL/I programs; they can be created, accessed, or modified by programs written in PL/I or other languages or by the utility programs of the operating system.

Your PL/I program recognizes and processes information in a data set by using a symbolic or logical representation of the data set called a *file*. This chapter describes how to associate data sets with the files known within your program. It introduces the five major types of data sets, how they are organized and accessed, and some of the file and data set characteristics you need to know how to specify.

Note: INDEXED is supported only under batch.

Associating data sets with files under OS/390

A file used within a PL/I program has a *PL/I file name*. The physical data set external to the program has a name by which it is known to the operating system: a *data set name* or *dsname*. In some cases the data set has no name; it is known to the system by the device on which it exists.

The operating system needs a way to recognize which physical data set is referred to by your program, so you must write a *data definition* or *DD* statement, external to your program, that associates the PL/I file name with a dsname. For example, if you have the following file declaration in your program:

```
DCL STOCK FILE STREAM INPUT;
```

you should create a DD statement with a *data definition name* (*ddname*) that matches the name of the PL/I file. The DD statement specifies a physical data set name (dsname) and gives its characteristics:

```
//GO.STOCK  DD DSN=PARTS.INSTOCK, . . .
```

You'll find some guidance in writing DD statements in this manual, but for more detail refer to the job control language (JCL) manuals for your system.

There is more than one way to associate a data set with a PL/I file. You associate a data set with a PL/I file by ensuring that the ddname of the DD statement that defines the data set is the same as one of the following:

- The declared PL/I file name
- The character-string value of the expression specified in the TITLE option of the associated OPEN statement.

You must choose your PL/I file names so that the corresponding ddnames conform to the following restrictions:

- If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that explicitly opens the file, the ddname defaults to the file name. If the file name is longer than 8 characters, the default ddname is composed of the first 8 characters of the file name.
- The character set of the JCL does not contain the break character (_). Consequently, this character cannot appear in ddnames. Do not use break

characters among the first 8 characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters $, @, and #, however, are valid for ddnames, but the first character must be one of the letters A through Z.

Since external names are limited to 7 characters, an external file name of more than 7 characters is shortened into a concatenation of the first 4 and the last 3 characters of the file name. Such a shortened name is **not**, however, the name used as the ddname in the associated DD statement.

Consider the following statements:

1. `OPEN FILE(MASTER);`
2. `OPEN FILE(OLDMASTER);`
3. `READ FILE(DETAIL) ...;`

When statement number 1 is run, the file name MASTER is taken to be the same as the ddname of a DD statement in the current job step. When statement number 2 is run, the name OLDMASTE is taken to be the same as the ddname of a DD statement in the current job step. (The first 8 characters of a file name form the ddname. If OLDMASTER is an external name, it will be shortened by the compiler to OLDMTER for use within the program.) If statement number 3 causes implicit opening of the file DETAIL, the name DETAIL is taken to be the same as the ddname of a DD statement in the current job step.

In each of the above cases, a corresponding DD statement must appear in the job stream; otherwise, the UNDEFINEDFILE condition is raised. The three DD statements could start as follows:

1. `//MASTER   DD ...`
2. `//OLDMASTE DD ...`
3. `//DETAIL   DD ...`

If the file reference in the statement which explicitly or implicitly opens the file is not a file constant, the DD statement name **must** be the same as the value of the file reference. The following example illustrates how a DD statement should be associated with the value of a file variable:

```
DCL PRICES FILE VARIABLE,
    RPRICE FILE;
      PRICES = RPRICE;
      OPEN FILE(PRICES);
```

The DD statement should associate the data set with the file constant RPRICE, which is the value of the file variable PRICES, thus:

```
//RPRICE DD DSNAME=...
```

Use of a file variable also allows you to manipulate a number of files at various times by a single statement. For example:

```
DECLARE F FILE VARIABLE,
        A FILE,
        B FILE,
        C FILE;
          .
          .
          .
     DO F=A,B,C;
        READ FILE (F) ...;
          .
          .
          .
     END;
```

The READ statement reads the three files A, B, and C, each of which can be associated with a different data set. The files A, B, and C remain open after the READ statement is executed in each instance.

The following OPEN statement illustrates use of the TITLE option:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

For this statement to be executed successfully, you must have a DD statement in the current job step with DETAIL1 as its ddname. It could start as follows:

```
//DETAIL1 DD DSNAME=DETAILA,...
```

Thus, you associate the data set DETAILA with the file DETAIL through the ddname DETAIL1.

Associating several files with one data set

You can use the TITLE option to associate two or more PL/I files with the same external data set at the same time. This is illustrated in the following example, where INVNTRY is the name of a DD statement defining a data set to be associated with two files:

```
OPEN FILE (FILE1) TITLE('INVNTRY');
OPEN FILE (FILE2) TITLE('INVNTRY');
```

If you do this, be careful. These two files access a common data set through separate control blocks and data buffers. When records are written to the data set from one file, the control information for the second file will not record that fact. Records written from the second file could then destroy records written from the first file. PL/I does not protect against data set damage that might occur. If the data set is extended, the extension is reflected only in the control blocks associated with the file that wrote the data; this can cause an abend when other files access the data set.

Associating several data sets with one file

The file name can, at different times, represent entirely different data sets. In the above example of the OPEN statement, the file DETAIL1 is associated with the data set named in the DSNAME parameter of the DD statement DETAIL1. If you closed and reopened the file, you could specify a different ddname in the TITLE option to associate the file with a different data set.

Use of the TITLE option allows you to choose dynamically, at open time, one among several data sets to be associated with a particular file name. Consider the following example:

```
DO IDENT='A','B','C';
   OPEN FILE(MASTER)
        TITLE('MASTER1'||IDENT);
     .
     .
     .
   CLOSE FILE(MASTER);
END;
```

In this example, when MASTER is opened during the first iteration of the do-group, the associated ddname is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the ddname. During the second iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the ddname MASTER1B. Similarly, during the final iteration of the do-group, MASTER is associated with the ddname MASTER1C.

Concatenating several data sets

For input only, you can concatenate two or more sequential or regional data sets (that is, link them so that they are processed as one continuous data set) by omitting the ddname from all but the first of the DD statements that describe them. For example, the following DD statements cause the data sets LIST1, LIST2, and LIST3 to be treated as a single data set for the duration of the job step in which the statements appear:

```
//GO.LIST DD DSNAME=LIST1,DISP=OLD
//        DD DSNAME=LIST2,DISP=OLD
//        DD DSNAME=LIST3,DISP=OLD
```

When read from a PL/I program, the concatenated data sets need not be on the same volume. You cannot process concatenated data sets backward.

Associating data sets with files under OS/390 UNIX

A file used within a PL/I program has a PL/I file name. A data set also has a name by which it is known to the operating system.

PL/I needs a way to recognize the data set(s) to which the PL/I files in your program refer, so you must provide an identification of the data set to be used, or allow PL/I to use a default identification.

You can identify the data set explicitly using either an environment variable or the TITLE option of the OPEN statement.

Using environment variables

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

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

```
declare MyFile stream output;

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

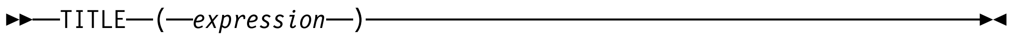
If you are familiar with the IBM mainframe environment, you can think of the environment variable much like you do the:

DD statement in OS/390
ALLOCATE statement in TSO

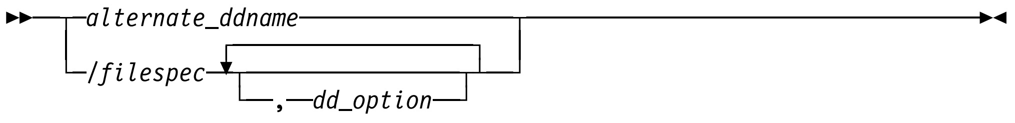
For more about the syntax and options you can use with the DD_DDNAME environment variable, see "Specifying characteristics using DD_DDNAME environment variables" on page 94.

Using the TITLE option of the OPEN statement

You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file, and, optionally, to provide additional characteristics of that data set.



The ***expression*** must yield a character string with the following syntax:



alternate_ddname
The name of an alternate DD_DDNAME environment variable. An alternate DD_DDNAME environment variable is 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 94.

Here is an example of using the OPEN statement in this manner:

```
open file(Payroll) title('/June.Dat,append(n),recsize(52)');
```

With this form, PL/I obtains all DD information either from the TITLE expression or from the ENVIRONMENT attribute of a file declaration. A DD_DDNAME environment variable is not referenced.

Attempting to use files not associated with data sets

If you attempt to use a file that has not been associated with a data set, (either through the use of the TITLE option of the OPEN statement or by establishing a DD_DDNAME environment variable), the UNDEFINEDFILE condition is raised. The only exceptions are the files SYSIN and SYSPRINT; these default to stdin and stdout, respectively.

How PL/I finds data sets

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

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

Specifying characteristics using DD_DDNAME environment variables

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

The syntax of the DD_DDNAME environment variable is:

```
                        ┌──────────────┐
►►──DD_DDNAME=filespec──▼┬────────────┬┴──────────────────────────────►◄
                         └─,──option──┘
```

Blanks are acceptable within the syntax. In addition, the syntax of the statement is not checked at the time the command is entered. It is verified when the data set is opened. If the syntax is wrong, UNDEFINEDFILE is raised with the oncode 96.

DD_DDNAME
Specifies the name of the environment variable. The DDNAME must be in upper case and can be either the name of a file constant or an alternate DDNAME that you specify in the TITLE option of your OPEN statement. The TITLE option is described in “Using the TITLE option of the OPEN statement” on page 93.

If you use an alternate DDNAME, and it is longer than 31 characters, only the first 31 characters are used in forming the environment variable name.

filespec
Specifies a file or the name of a device to be associated with the PL/I file.

option
The options you can specify as DD information.

The options that you can specify as DD information are described in the pages that follow, beginning with “APPEND” on page 95 and ending with “TYPE” on page 99.

APPEND

The APPEND option specifies whether an existing data set is to be extended or recreated.

```
           ┌Y┐
►►──APPEND──(──┴N┴──)─────────────────────────────────────────────►◄
```

Y Specifies that new records are to be added to the end of a sequential data set, or inserted in a relative or indexed data set.

N Specifies that, if the file exists, it is to be recreated.

The APPEND option applies only to OUTPUT files. APPEND is ignored if:

- The file does not exist
- The file does not have the OUTPUT attribute
- The organization is REGIONAL(1)

ASA

The ASA option applies to printer-destined files. This option specifies when the ANS control character in each record is to be interpreted.

```
        ┌N┐
►►──ASA──(──┴Y┴──)────────────────────────────────────────────────►◄
```

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

Y Specifies that the ANS print control characters are not to be translated; instead they are to be left as is for subsequent translation by a process you determine.

If the file is not a printer-destined file, the option is ignored.

BUFSIZE

The BUFSIZE option specifies the number of bytes for a buffer.

```
►►──BUFSIZE──(──n──)──────────────────────────────────────────────►◄
```

RECORD output is buffered by default and has a default value for BUFSIZE of 64k. STREAM output is buffered, but not by default, and has a default value for BUFSIZE of zero.

If the value of zero is given to BUFSIZE, the number of bytes for buffering is equal to the value specified in the RECSIZE or LRECL option.

The BUFSIZE option is valid only for a consecutive binary file. If the file is used for terminal input, you should assign the value of zero to BUFSIZE for increased efficiency.

CHARSET for record I/O

This version of the CHARSET option applies only to consecutive files using record I/O. It gives the user the capability of using ASCII data files as input files, and specifying the character set of output files.

```
             ┌─ASIS───┐
►►──CHARSET──(──┼─EBCDIC─┼──)──────────────────────────────────────────►◄
             └─ASCII──┘
```

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file have (output).

CHARSET for stream I/O

This version of the CHARSET option applies for stream input and output files. It gives the user the capability of using ASCII data files as input files, and specifying the character set of output files. If you attempt to specify ASIS when using stream I/O, no error is issued and character sets are treated as EBCDIC.

```
             ┌─EBCDIC─┐
►►──CHARSET──(──┴─ASCII──┴──)──────────────────────────────────────────►◄
```

Choose a suboption of CHARSET based on what form the file has (input) or what form you want the file to have (output).

DELAY

The DELAY option specifies the number of milliseconds to delay before retrying an operation that fails when a file or record lock cannot be obtained by the system.

```
           ┌─0─┐
►►──DELAY──(──┴─n─┴──)─────────────────────────────────────────────────►◄
```

This option is applicable only to 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.

```
             ┌─N─┐
►►──DELIMIT──(──┴─Y─┴──)───────────────────────────────────────────────►◄
```

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

LRECL

The LRECL option is the same as the RECSIZE option.

```
►►──LRECL──(──n──)────────────────────────────────────────────────►◄
```

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

LRMSKIP

The LRMSKIP option allows output to commence on the `nth` (`n` refers to the value specified with the SKIP option of the PUT or GET statement) line of the first page for the first SKIP format item to be executed after a file is opened.

```
               ┌─N─┐
►►──LRMSKIP──(─┴─Y─┴──)───────────────────────────────────────────►◄
```

If `n` is zero or 1, output commences on the first line of the first page.

PROMPT

The PROMPT option specifies whether or not colons should be visible as prompts for stream input from the terminal.

```
              ┌─N─┐
►►──PROMPT──(─┴─Y─┴──)────────────────────────────────────────────►◄
```

PUTPAGE

The PUTPAGE option specifies whether or not the form feed character should be followed by a carriage return character. This option 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─┐
►►──PUTPAGE──(─┴─CR───┴──)────────────────────────────────────────►◄
```

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

CR
: Indicates that the carriage return character is appended to the form feed character. This option should be specified if output is sent to non-IBM printers.

RECCOUNT

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

```
►►──RECCOUNT──(──n──)─────────────────────────────────────────────►◄
```

The RECCOUNT option is ignored if PL/I does not create, or recreate, the data set. 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.

```
              ┌─512─┐
►►──RECSIZE──(──┴─n───┴──)────────────────────────────────────────►◄
```

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

SAMELINE

The SAMELINE option specifies whether the system prompt occurs on the same line as the statement that prompts for input.

```
                ┌─N─┐
►►──SAMELINE──(──┴─Y─┴──)─────────────────────────────────────────►◄
```

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

Example 1
Given the statement `PUT SKIP LIST('ENTER:');`, output is as follows:

prompt(y), sameline(y)	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)

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.

```
             ┌─N─┐
►►──SKIP0──(──┴─Y─┴──)────────────────────────────────────────────►◄
```

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

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

The following example shows how you could make the output to the terminal skip zero lines so that the cursor moves to the beginning of the current output line:

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

TYPE

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

```
           ┌─LF──────┐
►►──TYPE──(─┼─CRLF────┼──)──────────────────────────────────────────────►◄
           ├─TEXT────┤
           ├─FIXED───┤
           ├─CRLFEOF─┤
           └─U───────┘
```

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

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

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

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

TEXT
Equivalent to LF.

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

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

CRLFEOF
Except for output files, this suboption specifies the same information as CRLF. When one of these files is closed for output, an end-of-file marker is appended to the last record.

U Indicates that records are unformatted. These unformatted files cannot be used by any record or stream I/O statements except OPEN and CLOSE. You can read from a TYPE(U) file only by using the FILEREAD built-in function. You can write to a TYPE(U) file only by using the FILEWRITE built-in function.

The TYPE option applies only to CONSECUTIVE files, except that it is ignored for printer-destined files with ASA(N) applied.

If your program attempts to access an existing data set with TYPE(FIXED) in effect and the length of the data set is not a multiple of the logical record length you specify, PL/I raises the UNDEFINEDFILE condition.

When using nonprint files with the TYPE(FIXED) attribute, SKIP is replaced by trailing blanks to the end of the line. If TYPE(LF) is being used, SKIP is replaced by LF with no trailing blanks.

Establishing data set characteristics

A data set consists of records stored in a particular format which the operating system data management routines understand. When you declare or open a file in your program, you are describing to PL/I and to the operating system the characteristics of the records that file will contain. You can also use JCL or an expression in the TITLE option of the OPEN statement to describe to the operating system the characteristics of the data in data sets or in the PL/I files associated with them.

You do not always need to describe your data both within the program and outside it; often one description will serve for both data sets and their associated PL/I files. There are, in fact, advantages to describing your data's characteristics in only one place. These are described later in this chapter and in following chapters.

To effectively describe your program data and the data sets you will be using, you need to understand something of how the operating system moves and stores data.

Blocks and records

The items of data in a data set are arranged in blocks separated by interblock gaps (IBG). (Some manuals refer to these as interrecord gaps.)

A *block* is the unit of data transmitted to and from a data set. Each block contains one record, part of a record, or several records. You can specify the block size in the BLKSIZE parameter of the DD statement or in the BLKSIZE option of the ENVIRONMENT attribute.

A *record* is the unit of data transmitted to and from a program. You can specify the record length in the LRECL parameter of the DD statement, in the TITLE option of the OPEN statement, or in the RECSIZE option of the ENVIRONMENT attribute.

When writing a PL/I program, you need consider only the records that you are reading or writing; but when you describe the data sets that your program will create or access, you must be aware of the relationship between blocks and records.

Blocking conserves storage space in a magnetic storage volume because it reduces the number of interblock gaps, and it can increase efficiency by reducing the number of input/output operations required to process a data set. Records are blocked and deblocked by the data management routines.

Information interchange codes: The normal code in which data is recorded is the Extended Binary Coded Decimal Interchange Code (EBCDIC).

Each character in the ASCII code is represented by a 7-bit pattern and there are 128 such patterns. The ASCII set includes a substitute character (the SUB control character) that is used to represent EBCDIC characters having no valid ASCII code. The ASCII substitute character is translated to the EBCDIC SUB character, which has the bit pattern 00111111.

Record formats

The records in a data set have one of the following formats:

Fixed-length
Variable-length
Undefined-length.

Records can be blocked if required. The operating system will deblock fixed-length and variable-length records, but you must provide code in your program to deblock undefined-length records.

You specify the record format in the RECFM parameter of the DD statement, in the TITLE option of the OPEN statement, or as an option of the ENVIRONMENT attribute.

Fixed-length records

You can specify the following formats for fixed-length records:

F	Fixed-length, unblocked
FB	Fixed-length, blocked

In a data set with fixed-length records, as shown in Figure 17, all records have the same length. If the records are blocked, each block usually contains an equal number of fixed-length records (although a block can be truncated). If the records are unblocked, each record constitutes a block.

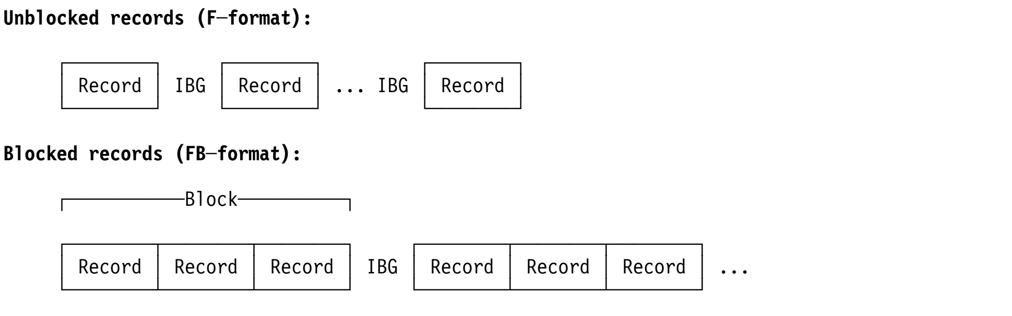


Figure 17. Fixed-length records

Because it bases blocking and deblocking on a constant record length, the operating system processes fixed-length records faster than variable-length records.

Variable-length records

You can specify the following formats for variable-length records:

V	Variable-length, unblocked
VB	Variable-length, blocked

V-format allows both variable-length records and variable-length blocks. A 4-byte prefix of each record and the first 4 bytes of each block contain control information for use by the operating system (including the length in bytes of the record or block). Because of these control fields, variable-length records cannot be read backward.

V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record. The first 4 bytes of the block contain block control information, and the next 4 contain record control information.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate. The first 4 bytes of the block contain block control information, and a 4-byte prefix of each record contains record control information.

Undefined-length records

U-format allows the processing of records that do not conform to F- and V-formats. The operating system and the compiler treat each block as a record; your program must perform any required blocking or deblocking.

Data set organization

The data management routines of the operating system can handle a number of types of data sets, which differ in the way data is stored within them and in the allowed means of access to the data. The three main types of non-VSAM data sets and the corresponding keywords describing their PL/I organization[1] are as follows:

Type of data set	**PL/I organization**
Sequential	CONSECUTIVE or ORGANIZATION(consecutive)
Indexed	INDEXED or ORGANIZATION(indexed)
Direct	REGIONAL or ORGANIZATION(relative)

A fourth type, *partitioned*, has no corresponding PL/I organization.

PL/I also provides support for three types of VSAM data organization: *ESDS*, *KSDS*, and *RRDS*. For more information about VSAM data sets, see Chapter 10, “Defining and using VSAM data sets” on page 179.

In a *sequential* (or CONSECUTIVE) data set, records are placed in physical sequence. Given one record, the location of the next record is determined by its physical position in the data set. Sequential organization can be selected for direct-access devices.

[1] Do not confuse the terms “sequential” and “direct” with the PL/I file attributes SEQUENTIAL and DIRECT. The attributes refer to how the file is to be processed, and not to the way the corresponding data set is organized.

An *indexed sequential* (or INDEXED) data set must reside on a direct-access volume. An index or set of indexes maintained by the operating system gives the location of certain principal records. This allows direct retrieval, replacement, addition, and deletion of records, as well as sequential processing.

A *direct* (or REGIONAL) data set must reside on a direct-access volume. The data set is divided into regions, each of which contains one or more records. A key that specifies the region number allows direct-access to any record; sequential processing is also possible.

In a *partitioned* data set, independent groups of sequentially organized data, each called a member, reside in a direct-access data set. The data set includes a directory that lists the location of each member. Partitioned data sets are often called *libraries*. The compiler includes no special facilities for creating and accessing partitioned data sets. Each member can be processed as a CONSECUTIVE data set by a PL/I program. The use of partitioned data sets as libraries is described under Chapter 6, “Using libraries” on page 117.

Labels

The operating system uses internal labels to identify direct-access volumes and to store data set attributes (for example, record length and block size). The attribute information must originally come from a DD statement or from your program.

IBM standard labels have two parts: the initial volume label and header labels. The initial volume label identifies a volume and its owner; the header labels precede and follow each data set on the volume. Header labels contain system information, device-dependent information (for example, recording technique), and data-set characteristics.

Direct-access volumes have IBM standard labels. Each volume is identified by a volume label, which is stored on the volume. This label contains a volume serial number and the address of a volume table of contents (VTOC). The table of contents, in turn, contains a label, termed a *data set control block* (DSCB), for each data set stored on the volume.

Data Definition (DD) statement

A data definition (DD) statement is a job control statement that defines a data set to the operating system, and is a request to the operating system for the allocation of input/output resources. If the data sets are not dynamically allocated, each job step must include a DD statement for each data set that is processed by the step.

Your *OS/390 JCL User's Guide* describes the syntax of job control statements. The operand field of the DD statement can contain keyword parameters that describe the location of the data set (for example, volume serial number and identification of the unit on which the volume will be mounted) and the attributes of the data itself (for example, record format).

The DD statement enables you to write PL/I source programs that are independent of the data sets and input/output devices they will use. You can modify the parameters of a data set or process different data sets without recompiling your program.

The following paragraphs describe the relationship of some operands of the DD statement to your PL/I program.

Write validity checking, which was standard in PL/I Version 1, is no longer performed. Write validity checking can be requested through the OPTCD subparameter of the DCB parameter of the JCL DD statement. See *OS/VS2 Job Control Language* manual.

Use of the conditional subparameters

If you use the conditional subparameters of the DISP parameter for data sets processed by PL/I programs, the step abend facility must be used. The step abend facility is obtained as follows:

1. The ERROR condition should be raised or signaled whenever the program is to terminate execution after a failure that requires the application of the conditional subparameters.
2. The PL/I user exit must be changed to request an ABEND.

Data set characteristics

The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a data set, and the way it will be processed, at run time. Whereas the other parameters of the DD statement deal chiefly with the identity, location, and disposal of the data set, the DCB parameter specifies information required for the processing of the records themselves. The subparameters of the DCB parameter are described in your *OS/390 JCL User's Guide*.

The DCB parameter contains subparameters that describe:

- The organization of the data set and how it will be accessed (CYLOFL, DSORG, LIMCT, NTM, and OPTCD subparameters)
- Device-dependent information such as the line spacing for a printer (CODE, FUNC, MODE, OPTCD=J, PRTSP, and STACK subparameters)
- The record format (BLKSIZE, KEYLEN, LRECL, and RECFM subparameters)
- The ASA control characters (if any) that will be inserted in the first byte of each record (RECFM subparameter).

You can specify BLKSIZE, LRECL, KEYLEN, and RECFM (or their equivalents) in the ENVIRONMENT attribute of a file declaration in your PL/I program instead of in the DCB parameter.

You cannot use the DCB parameter to override information already established for the data set in your PL/I program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied are ignored.

An example of the DCB parameter is:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

which specifies that fixed-length records, 40 bytes in length, are to be grouped together in a block 400 bytes long.

OS/390 UNIX Only

Specifying characteristics using DD_DDNAME environment variables

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

The syntax of the DD_DDNAME environment variable is:

```
                         ┌─────────────┐
►►──DD_DDNAME=filespec───▼─────────────┴────────────────────────►◄
                          └─,──option──┘
```

Note: The option list must be enclosed by quotes when any of the suboptions are specified.

Blanks are acceptable within the syntax. In addition, the syntax of the statement is not checked at the time the command is entered. It is verified when the data set is opened. If the syntax is wrong, UNDEFINEDFILE is raised with the ONCODE 96.

DD_DDNAME
Specifies the name of the environment variable. The DDNAME must be in upper case and can be either the name of a file constant or an alternate DDNAME that you specify in the TITLE option of your OPEN statement.

If you use an alternate DDNAME, and it is longer than 31 characters, only the first 31 characters are used in forming the environment variable name.

filespec
Specifies a file or the name of a device to be associated with the PL/I file.

option
The options you can specify as DD information are:

APPEND	DELAY	RECOUNT
ASA	LRECL	RECSIZE
BUFSIZE	PROMPT	SKIP0
CHARSET	PUTPAGE	TYPE

End of OS/390 UNIX Only

Using the TITLE option of the OPEN statement

You can use the TITLE option of the OPEN statement to identify the data set to be associated with a PL/I file and, optionally, to provide additional characteristics of the data set.

```
►►──TITLE──(──expression──)──────────────────────────────────►◄
```

The ***expression*** must yield a character string with the following syntax:

```
►►──┬─alternate_ddname────────────────────┬─────────────────────►◄
    │ ┌─────────────────────────────────┐ │
    └─▼─/filespec──┬───┬──┬───────────┬─┴─┘
                   └─,─┘  └─dd_option─┘
```

alternate_ddname

The name of an alternate DD_DDNAME environment variable. An alternate DD_DDNAME environment variable is one not named after a file constant. For example, if you had a file named INVENTRY in your program, and you establish two DD_DDNAME environment variables—the first named INVENTRY and the second named PARTS—you could associate the file with the second one using this statement:

```
open file(Inventry) title('PARTS');
```

filespec

Any valid OS/390 UNIX or OS/390 PDS file specification.

dd_option

One or more options allowed in a DD_DDNAME environment variable. For more information about options of the DD_DDNAME variable, see "Specifying characteristics using DD_DDNAME environment variables" on page 105.

Here is an example of using the OPEN statement in this manner:

```
open file(Payroll) title('/June.Dat, append(n),recsize(52)');
```

With this form, PL/I obtains all DD information either from the TITLE expression or from the ENVIRONMENT attribute of a file declaration. A DD_DDNAME environment variable is not referenced.

Associating PL/I files with data sets

Opening a file: The execution of a PL/I OPEN statement associates a file with a data set. This requires merging of the information describing the file and the data set. If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Subroutines of the PL/I library create a skeleton data control block for the data set. They use the file attributes from the DECLARE and OPEN statements and any attributes implied by the declared attributes, to complete the data control block as far as possible. (See Figure 18 on page 107.) They then issue an OPEN macro instruction, which calls the data management routines to check that the correct volume is mounted and to complete the data control block.

The data management routines examine the data control block to see what information is still needed and then look for this information, first in the DD statement, and finally, if the data set exists and has standard labels, in the data set labels. For new data sets, the data management routines begin to create the labels (if they are required) and to fill them with information from the data control block.

Neither the DD statement nor the data set label can override information provided by the PL/I program; nor can the data set label override information provided by the DD statement.

When the DCB fields are filled in from these sources, control returns to the PL/I library subroutines. If any fields still are not filled in, the PL/I OPEN subroutine provides default information for some of them. For example, if LRECL is not specified, it is provided from the value given for BLKSIZE.

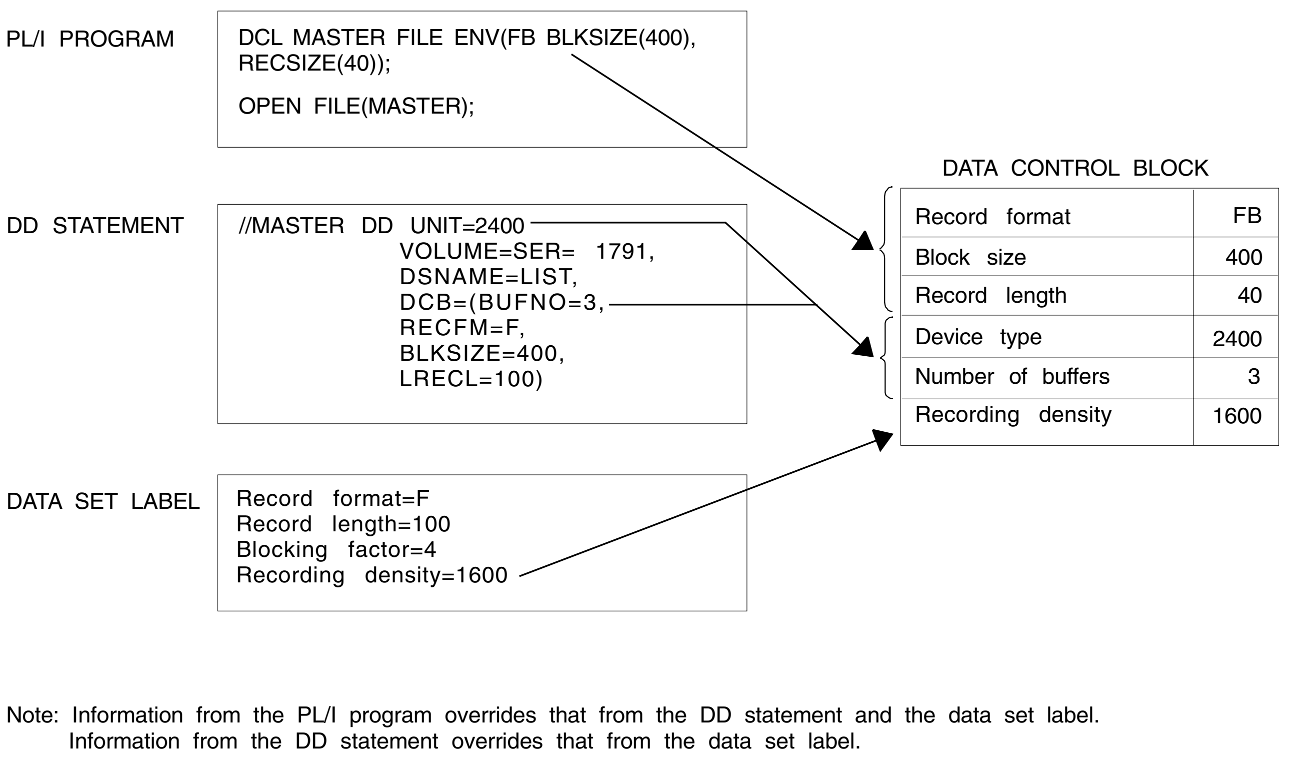


Figure 18. How the operating system completes the DCB

Closing a file: The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated. The PL/I library subroutines first issue a CLOSE macro instruction and, when control returns from the data management routines, release the data control block that was created when the file was opened. The data management routines complete the writing of labels for new data sets and update the labels of existing data sets.

Specifying characteristics in the ENVIRONMENT attribute

You can use various options in the ENVIRONMENT attribute. Each type of file has different attributes and environment options, which are listed below.

The ENVIRONMENT attribute: You use the ENVIRONMENT attribute of a PL/I file declaration file to specify information about the physical organization of the data set associated with a file, and other related information. The format of this information must be a parenthesized option list.

```
▶▶──ENVIRONMENT──(──option-list──)──────────────────────────────────────▶◀
```

Abbreviation: ENV

You can specify the options in any order, separated by blanks or commas.

The following example illustrates the syntax of the ENVIRONMENT attribute in the context of a complete file declaration (the options specified are for VSAM and are discussed in Chapter 10, “Defining and using VSAM data sets” on page 179).

```
DCL FILENAME FILE RECORD SEQUENTIAL
    INPUT ENV(VSAM GENKEY);
```

Table 10 summarizes the ENVIRONMENT options and file attributes. Certain qualifications on their use are presented in the notes and comments for the figure. Those options that apply to more than one data set organization are described in the remainder of this chapter. In addition, in the following chapters, each option is described with each data set organization to which it applies.

Table 10. Attributes of PL/I file declarations

Data set type	**Stream**	**Record**										
		Sequential							**Direct**			**Legend:** **C Checked for VSAM** **D Default** **I Must be specified or implied** **N Ignored for VSAM** **O Optional** **S Must be specified** **- Invalid**
		Consecutive		**Regional**								
File Type	**Consecutive**	**Buffered**	**Unbuffered**	**Buffered**	**Unbuffered**	**Teleprocessing**	**Indexed**	**VSAM**	**Regional**	**Indexed**	**VSAM**	
File attributes[1]												**Attributes implied**
File	I	I	I	I	I	I	I	I	I	I	I	
Input[1]	D	D	D	D	D	D	D	D	D	D	D	File
Output	O	O	O	O	O	O	O	O	O	O	O	File
Environment	I	I	I	S	S	S	S	S	S	S	S	File
Stream	D	-	-	-	-	-	-	-	-	-	-	File
Print[1]	O	-	-	-	-	-	-	-	-	-	-	File stream output
Record	-	I	I	I	I	I	I	I	I	I	I	File
Update	-	O	O	O	O	-	O	O	O	O	O	File record
Sequential	-	D	D	D	D	-	D	D	-	-	D	File record
Buffered	-	D	-	D	-	I	D	D	-	-	S	File record
Keyed[2]	-	-	-	O	O	I	O	O	I	I	O	File record
Direct	-	-	-	-	-	-	-	S	S	S	S	File record keyed
ENVIRONMENT options												**Comments**
F|FB|V| VB||U	I	S	S	-	-	-	-	N	-	-	N	
F|FB|U	S	S	-	-	-	-	-	N	-	-	N	ASCII data sets only
F|V|U	-	-	-	S	S	-	-	N	S	-	N	Only F for REGIONAL(1)
F|FB|V|VB	-	-	-	-	-	-	S	N	-	S	N	
RECSIZE(n)	I	I	I	I	I	S	I	C	I	I	C	RECSIZE and/or BLKSIZE must be specified
BLKSIZE(n)	I	I	I	I	I	-	I	N	I	I	N	for consecutive, indexed, and regional files
SCALARVARYING	-	O	O	O	O	-	O	O	O	O	O	Invalid for ASCII data sets
CONSECUTIVE	D	D	D	-	-	-	-	O	-	-	O	Allowed for VSAM ESDS
CTLASA|CTL360	-	O	O	-	-	-	-	-	-	-	-	Invalid for ASCII data sets
GRAPHIC	O	-	-	-	-	-	-	-	-	-	-	
INDEXED	-	-	-	-	-	-	S	O	-	S	O	Allowed for VSAM ESDS
KEYLOC(n)	-	-	-	-	-	-	O	-	-	O	-	
ORGANIZATION	D			-	-	-			-			
GENKEY	-	-	-	-	-	-	O	O	-	O	O	INPUT or UPDATE files only; KEYED is required
REGIONAL(1)	-	-	-	S	S	-	-	-	S	-	-	
VSAM	-	-	-	-	-	-	-	S	-	-	S	
BKWD	-	-	-	-	-	-	-	O	-	-	O	
REUSE	-	-	-	-	-	-	-	O	-	-	O	OUTPUT file only

Notes:

1. A file with the INPUT attribute cannot have the PRINT attribute.
2. Keyed is required for INDEXED and REGIONAL output.

Data set organization options: The options that specify data set organization are:

```
►►──┬─CONSECUTIVE───────┬────────────────────────────────────────────►◄
    ├─INDEXED───────────┤
    ├─REGIONAL──(──1──)─┤
    └─VSAM──────────────┘
```

Each option is described in the discussion of the data set organization to which it applies.

Other ENVIRONMENT options: You can use a constant or variable with those ENVIRONMENT options that require integer arguments, such as block sizes and record lengths. The variable must not be subscripted or qualified, and must have attributes FIXED BINARY(31,0) and STATIC.

The list of equivalents for ENVIRONMENT options and DCB parameters are:

ENVIRONMENT option	**DCB subparameter**
Record format	RECFM[1]
RECSIZE	LRECL
BLKSIZE	BLKSIZE
CTLASA|CTL360	RECFM
KEYLENGTH	KEYLEN

Record formats for record-oriented data transmission: Record formats supported depend on the data set organization.

```
►►──┬─F──┬───────────────────────────────────────────────────────────►◄
    ├─FB─┤
    ├─V──┤
    ├─VB─┤
    └─U──┘
```

Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
Variable-length	V	unblocked
	VB	blocked
		unblocked, ASCII
		blocked, ASCII
Undefined-length	U	(cannot be blocked)

When U-format records are read into a varying-length string, PL/I sets the length of the string to the block length of the retrieved data.

These record format options do not apply to VSAM data sets. If you specify a record format option for a file associated with a VSAM data set, the option is ignored.

Record formats for stream-oriented data transmission: The record format options for stream-oriented data transmission are discussed in "Using stream-oriented data transmission" on page 123.

RECSIZE option: The RECSIZE option specifies the record length.

```
►►──RECSIZE──(──record-length──)──────────────────────────────────────►◄
```

For files associated with VSAM data sets, **record-length** is the sum of:

1. The length required for data. For variable-length and undefined-length records, this is the maximum length.
2. Any control bytes required. Variable-length records require 4 (for the record-length prefix); fixed-length and undefined-length records do not require any.

For VSAM data sets, the maximum and average lengths of the records are specified to the Access Method Services utility when the data set is defined. If you include the RECSIZE option in the file declaration for checking purposes, you should specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

You can specify **record-length** as an integer or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:
: Fixed-length, and undefined (except ASCII data sets): 32760

  V-format: 32756

  ASCII data sets: 9999

  VSAM data sets: 32761

Zero value:
: A search for a valid value is made *first*:

  - In the DD statement for the data set associated with the file, and *second*
  - In the data set label.

  If neither of these provides a value, default action is taken (see “Record format, BLKSIZE, and RECSIZE defaults” on page 111).

Negative Value:
: The UNDEFINEDFILE condition is raised.

BLKSIZE option: The BLKSIZE option specifies the maximum block size on the data set.

```
►►──BLKSIZE──(──block-size──)─────────────────────────────────────────►◄
```

block-size is the sum of:

1. The total length(s) of one of the following:
   - A single record
   - A single record and either one or two record segments
   - Several records
   - Several records and either one or two record segments
   - Two record segments

- A single record segment.

 For variable-length records, the length of each record or record segment includes the 4 control bytes for the record or segment length.

 The above list summarizes all the possible combinations of records and record segments options: fixed- or variable-length blocked or unblocked

2. Any further control bytes required.
 - Variable-length blocked records require 4 (for the block size).
 - Fixed-length and undefined-length records do not require any further control bytes.
3. Any block prefix bytes required (ASCII data sets only).

block-size can be specified as an integer, or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:
: 32760

Zero value:
: If you set BLKSIZE to 0, under OS/390 the Data Facility Product sets the block size. For an elaboration of this topic, see “Record format, BLKSIZE, and RECSIZE defaults.” BLKSIZE defaults.

Negative value:
: The UNDEFINEDFILE condition is raised.

The relationship of block size to record length depends on the record format:

FB-format
: The block size must be a multiple of the record length.

VB-format:
: The block size must be equal to or greater than the sum of:
 1. The maximum length of any record
 2. Four control bytes.

Notes:

- Use the BLKSIZE option with unblocked (F- or V-format) records in either of the following ways:
 - Specify the BLKSIZE option, but not the RECSIZE option. Set the record length equal to the block size (minus any control or prefix bytes), and leave the record format unchanged.
 - Specify both BLKSIZE and RECSIZE and ensure that the relationship of the two values is compatible with blocking for the record format you use. Set the record format to FB or VB, whichever is appropriate.
- The BLKSIZE option does not apply to VSAM data sets, and is ignored if you specify it for one.

Record format, BLKSIZE, and RECSIZE defaults: If you do not specify either the record format, block size, or record length for a non-VSAM data set, the following default action is taken:

Record format:
A search is made in the associated DD statement or data set label. If the search does not provide a value, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets or the foreground terminal, in which case the record format is set to U.

Block size or record length:
If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, and if this value is incompatible with the value in the specified option, the UNDEFINEDFILE condition is raised. If the search is unsuccessful, a value is derived from the specified option (with the addition or subtraction of any control or prefix bytes).

If neither is specified, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets, in which case BLKSIZE is set to 121 for F-format or U-format records and to 129 for V-format records. For files associated with the foreground terminal, RECSIZE is set to 120.

If you are using OS/390 with the Data Facility Product system-determined block size, DFP determines the optimum block size for the device type assigned. If you specify BLKSIZE(0) in either the DD assignment or the ENVIRONMENT statement, DFP calculates BLKSIZE using the record length, record format, and device type.

GENKEY option — key classification: The GENKEY (generic key) option applies only to INDEXED and VSAM key-sequenced data sets. It enables you to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

```
►►──GENKEY──────────────────────────────────────────────────────────►◄
```

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

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an INDEXED data set from the first nondummy record that has a key in a particular class. You identify the class by including its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although you can retrieve the first record having a key in a particular class by using a READ with the KEY option, you cannot obtain the actual key unless the records have embedded keys, since the KEYTO option cannot be used in the same statement as the KEY option.

In the following example, a key length of more than 3 bytes is assumed:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (GENKEY);
          .
          .
          .
       READ FILE(IND) INTO(INFIELD)
                      KEY ('ABC');
          .
          .
          .
NEXT:  READ FILE (IND) INTO (INFIELD);
          .
          .
          .
       GO TO NEXT;
```

The first READ statement causes the first nondummy record in the data set whose key begins with “ABC” to be read into INFIELD; each time the second READ statement is executed, the nondummy record with the next higher key is retrieved. Repeated execution of the second READ statement could result in reading records from higher key classes, since no indication is given when the end of a key class is reached. It is your responsibility to check each key if you do not wish to read beyond the key class. Any subsequent execution of the first READ statement would reposition the file to the first record of the key class “ABC”.

If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

The presence or absence of the GENKEY option affects the execution of a READ statement which supplies a source key that is shorter than the key length specified in the KEYLEN subparameter. This KEYLEN subparameter is found in the DD statement that defines the indexed data set. If you specify the GENKEY option, it causes the source key to be interpreted as a generic key, and the data set is positioned to the first nondummy record in the data set whose key begins with the source key. If you do not specify the GENKEY option, a READ statement's short source key is padded on the right with blanks to the specified key length, and the data set is positioned to the record that has this padded key (if such a record exists). For a WRITE statement, a short source key is always padded with blanks.

Use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered to be the only member of its class).

SCALARVARYING option — varying-length strings: You use the SCALARVARYING option in the input/output of varying-length strings; you can use it with records of any format.

```
►►──SCALARVARYING──────────────────────────────────────────────────►◄
```

When storage is allocated for a varying-length string, the compiler includes a 2-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if SCALARVARYING is specified for the file.

When you use locate mode statements (LOCATE and READ SET) to create and read a data set with element varying-length strings, you must specify SCALARVARYING to indicate that a length prefix is present, since the pointer that locates the buffer is always assumed to point to the start of the length prefix.

When you specify SCALARVARYING and element varying-length strings are transmitted, you must allow two bytes in the record length to include the length prefix.

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

You must not specify SCALARVARYING and CTLASA/CTL360 for the same file, as this causes the first data byte to be ambiguous.

KEYLENGTH option: Use the KEYLENGTH option to specify the length of the recorded key for KEYED files where `n` is the length. You can specify KEYLENGTH for INDEXED files.

```
►►──KEYLENGTH──(──n──)──────────────────────────────────────────────►◄
```

If you include the KEYLENGTH option in a VSAM file declaration for checking purposes, and the key length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

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

```
                    ┌─CONSECUTIVE─┐
►►──ORGANIZATION──(─┼─INDEXED─────┼─)──────────────────────────────►◄
                    └─RELATIVE────┘
```

CONSECUTIVE
Specifies that the files is associated with a consecutive data set. A consecutive file can be either a native data set or a VSAM, ESDS, RRDS, or KSDS data set.

RELATIVE
Specifies that the file is associated with a relative data set. RELATIVE specifies that the data set contains records that do not have recorded keys. A relative file is a VSAM direct data set. Relative keys range from `1` to `nnnn`.

Data set types used by PL/I record I/O

Data sets with the RECORD attribute are processed by record-oriented data transmission in which data is transmitted to and from auxiliary storage exactly as it appears in the program variables; no data conversion takes place. A record in a data set corresponds to a variable in the program.

Table 11 on page 115 shows the facilities that are available with the various types of data sets that can be used with PL/I Record I/O.

Table 11. A comparison of data set types available to PL/I record I/O

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
SEQUENCE	Key order	Entry order	Num-bered	Key order	Entry order	By region
DEVICES	DASD	DASD	DASD	DASD	DASD, card, etc.	DASD
ACCESS 1 By key 2 Sequential	123	123	123	12	2	12
Alternate index access as above	123	123	No	No	No	No
How extended	With new keys	At end	In empty slots	With new keys	At end	In empty slots
DELETION 1 Space reusable 2 Space not reusable	Yes, 1	No	Yes, 1	Yes, 2	No	Yes, 2

The following chapters describe how to use Record I/O data sets for different types of data sets:

- Chapter 7, “Defining and using consecutive data sets” on page 123
- Chapter 8, “Defining and using indexed data sets” on page 147
- Chapter 9, “Defining and using regional data sets” on page 168
- Chapter 10, “Defining and using VSAM data sets” on page 179

OS/390 UNIX System Services Only

Setting environment variables

There are a number of environment variables that can be set and exported for use with OS/390 UNIX.

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

The following example illustrates how to set environment variables:

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

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

You can use the ECHO command to determine the current setting of an environment variable. To define the value of BYPASS, you can use either of the following two examples:

```
echo $LANG

echo $LIBPATH
```

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. Environment variables are discussed above in “Setting environment variables” on page 115.

Redirecting standard input, output, and error devices

You can also redirect standard input, standard output, and standard error devices to a file. You could use redirection in the following program:

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

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

```
hello2 > hello2.out
```

If you want to combine stdout and stderr in a single file, enter the following command:

```
hello2 > hello2.out 2>&1
```

As is true with display statements, the *greater than* sign redirects the output to the file that is specified after it, in this case `hello2.out`. This means that the word 'Hello' is written in the file `hello2.out`. Note also that the output includes printer control characters since the PRINT attribute is applied to SYSPRINT by default.

READ statements can access data from `stdin`; however, the record into which the data is to be put must have an LRECL equal to 1.

End of OS/390 UNIX System Services Only

Chapter 6. Using libraries

Within the OS/390 operating system, the terms “partitioned data set,” “partitioned data set/extension,” and “library” are synonymous and refer to a type of data set that can be used for the storage of other data sets (usually programs in the form of source, object or load modules). A library must be stored on direct-access storage and be wholly contained in one volume. It contains independent, consecutively organized data sets, called members. Each member has a unique name, not more than 8 characters long, which is stored in a directory that is part of the library. All the members of one library must have the same data characteristics because only one data set label is maintained.

You can create members individually until there is insufficient space left for a new entry in the directory, or until there is insufficient space for the member itself. You can access members individually by specifying the member name.

Use DD statements or their conversational mode equivalent to create and access members.

You can delete members by means of the IBM utility program IEHPROGM. This deletes the member name from the directory so that the member can no longer be accessed, but you cannot use the space occupied by the member itself again unless you recreate the library or compress the unused space using, for example, the IBM utility program IEBCOPY. If you attempt to delete a member by using the DISP parameter of a DD statement, it causes the whole data set to be deleted.

Types of libraries

You can use the following types of libraries with a PL/I program:

- The system program library SYS1.LINKLIB or its equivalent. This can contain all system processing programs such as compilers and the linkage editor.
- Private program libraries. These usually contain user-written programs. It is often convenient to create a temporary private library to store the load module output from the linkage editor until it is executed by a later job step in the same job. The temporary library will be deleted at the end of the job. Private libraries are also used for automatic library call by the linkage editor and the loader.
- The system procedure library SYS1.PROCLIB or its equivalent. This contains the job control procedures that have been cataloged for your installation.

How to use a library

A PL/I program can use a library directly. If you are adding a new member to a library, its directory entry will be made by the operating system when the associated file is closed, using the member name specified as part of the data set name.

If you are accessing a member of a library, its directory entry can be found by the operating system from the member name that you specify as part of the data set name.

More than one member of the same library can be processed by the same PL/I program, but only one such output file can be open at any one time. You access different members by giving the member name in a DD statement.

Creating a library

To create a library include in your job step a DD statement containing the information given in Table 12. The information required is similar to that for a consecutively organized data set (see “Defining files using record I/O” on page 139) except for the SPACE parameter.

Table 12. Information required when creating a library

Information Required	Parameter of DD statement
Type of device that will be used	UNIT=
Serial number of the volume that will contain the library	VOLUME=SER
Name of the library	DSNAME=
Amount of space required for the library	SPACE=
Disposition of the library	DISP=

SPACE parameter

The SPACE parameter in a DD statement that defines a library must always be of the form:

```
SPACE=(units,(quantity,increment,directory))
```

Although you can omit the third term (increment), indicating its absence by a comma, the last term, specifying the number of directory blocks to be allocated, must always be present.

The amount of auxiliary storage required for a library depends on the number and sizes of the members to be stored in it and on how often members will be added or replaced. (Space occupied by deleted members is not released.) The number of directory blocks required depends on the number of members and the number of aliases. You can specify an incremental quantity in the SPACE parameter that allows the operating system to obtain more space for the data set, if such is necessary at the time of creation or at the time a new member is added; the number of directory blocks, however, is fixed at the time of creation and cannot be increased.

For example, the DD statement:

```
//   PDS DD UNIT=SYSDA,VOL=SER=3412,
//   DSNAME=ALIB,
//   SPACE=(CYL,(5,,10)),
//   DISP=(,CATLG)
```

requests the job scheduler to allocate 5 cylinders of the DASD with a volume serial number 3412 for a new library name ALIB, and to enter this name in the system catalog. The last term of the SPACE parameter requests that part of the space allocated to the data set be reserved for ten directory blocks.

Creating and updating a library member

The members of a library must have identical characteristics. Otherwise, you might later have difficulty retrieving them. Identical characteristics are necessary because the volume table of contents (VTOC) will contain only one data set control block (DSCB) for the library and not one for each member. When using a PL/I program to create a member, the operating system creates the directory entry; you cannot place information in the user data field.

When creating a library and a member at the same time, your DD statement must include all the parameters listed under “Creating a library” on page 118 (although you can omit the DISP parameter if the data set is to be temporary). The DSNAME parameter must include the member name in parentheses. For example, DSNAME=ALIB(MEM1) names the member MEM1 in the data set ALIB. If the member is placed in the library by the linkage editor, you can use the linkage editor NAME statement or the NAME compile-time option instead of including the member name in the DSNAME parameter. You must also describe the characteristics of the member (record format, etc.) either in the DCB parameter or in your PL/I program. These characteristics will also apply to other members added to the data set.

When creating a member to be added to an existing library, you do not need the SPACE parameter. The original space allocation applies to the whole of the library and not to an individual member. Furthermore, you do not need to describe the characteristics of the member, since these are already recorded in the DSCB for the library.

To add two more members to a library in one job step, you must include a DD statement for each member, and you must close one file that refers to the library before you open another.

Examples

The use of the cataloged procedure IBMZC to compile a simple PL/I program and place the object module in a new library named EXLIB is shown in Figure 19 on page 120. The DD statement that defines the new library and names the object module overrides the DD statement SYSLIN in the cataloged procedure. (The PL/I program is a function procedure that, given two values in the form of the character string produced by the TIME built-in function, returns the difference in milliseconds.)

The use of the cataloged procedure IBMZCL to compile and link-edit a PL/I program and place the load module in the existing library HPU8.CCLM is shown in Figure 20 on page 120.

```
//OPT10#1 JOB
//TR       EXEC  IBMZC
//PLI.SYSLIN  DD  UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//      SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSIN   DD  *
   ELAPSE:  PROC(TIME1,TIME2);
     DCL (TIME1,TIME2) CHAR(9),
         H1 PIC '99' DEF TIME1,
         M1 PIC '99' DEF TIME1 POS(3),
         MS1 PIC '99999' DEF TIME1 POS(5),
         H2 PIC '99' DEF TIME2,
         M2 PIC '99' DEF TIME2 POS(3),
         MS2 PIC '99999' DEF TIME2 POS(5),
         ETIME FIXED DEC(7);
     IF H2<H1 THEN H2=H2+24;
     ETIME=((H2*60+M2)*60000+MS2)-((H1*60+M1)*60000+MS1);
     RETURN(ETIME);
   END ELAPSE;
/*
```

Figure 19. Creating new libraries for compiled object modules

```
//OPT10#2  JOB
//TRLE      EXEC  IBMZCL
//PLI.SYSIN   DD  *
   MNAME:  PROC  OPTIONS(MAIN);
     .
     .
     .
     program
     .
     .
     .

   END MNAME;
/*
//LKED.SYSLMOD  DD  DSNAME=HPU8.CCLM(DIRLIST),DISP=OLD
```

Figure 20. Placing a load module in an existing library

To use a PL/I program to add or delete one or more records within a member of a library, you must rewrite the entire member in another part of the library. This is rarely an economic proposition, since the space originally occupied by the member cannot be used again. You must use two files in your PL/I program, but both can be associated with the same DD statement. The program shown in Figure 22 on page 121 updates the member created by the program in Figure 21 on page 121. It copies all the records of the original member except those that contain only blanks.

```
//OPT10#3  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  NMEM:  PROC OPTIONS(MAIN);
     DCL IN FILE RECORD SEQUENTIAL INPUT,
         OUT FILE RECORD SEQUENTIAL OUTPUT,
         P POINTER,
         IOFIELD CHAR(80) BASED(P),
         EOF BIT(1) INIT('0'B);
     OPEN FILE(IN),FILE (OUT);
     ON ENDFILE(IN) EOF='1'B;
     READ FILE(IN) SET(P);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
     WRITE FILE(OUT) FROM(IOFIELD);
     READ FILE(IN) SET(P);
     END;
     CLOSE FILE(IN),FILE(OUT);
  END NMEM;
/*
//GO.OUT   DD  UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//     DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//     DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
  MEM:  PROC OPTIONS(MAIN);
        /* this is an incomplete dummy library member */
```

Figure 21. Creating a library member in a PL/I program

```
//OPT10#4  JOB
//TREX  EXEC IBMZCBG
//PLI.SYSIN      DD *
  UPDTM: PROC OPTIONS(MAIN);
     DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
         EOF BIT(1) INIT('0'B),
         DATA CHAR(80);
     ON ENDFILE(OLD)  EOF = '1'B;
     OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');
     READ FILE(OLD) INTO(DATA);
     DO WHILE (¬EOF);
     PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
     IF DATA=' ' THEN ;
     ELSE WRITE FILE(NEW) FROM(DATA);
     READ FILE(OLD) INTO(DATA);
     END;
   CLOSE FILE(OLD),FILE(NEW);
  END UPDTM;
/*
//GO.OLD DD  DSNAME=HPU8.ALIB(NMEM),DISP=(OLD,KEEP)
```

Figure 22. Updating a library member

Extracting information from a library directory

The directory of a library is a series of records (entries) at the beginning of the data set. There is at least one directory entry for each member. Each entry contains a member name, the relative address of the member within the library, and a variable amount of user data.

User data is information inserted by the program that created the member. An entry that refers to a member (load module) written by the linkage editor includes user data in a standard format, described in the systems manuals.

If you use a PL/I program to create a member, the operating system creates the directory entry for you and you cannot write any user data. However, you can use assembler language macro instructions to create a member and write your own user data. The method for using macro instructions to do this is described in the data management manuals.

Chapter 7. Defining and using consecutive data sets

This chapter covers consecutive data set organization and the ENVIRONMENT options that define consecutive data sets for stream and record-oriented data transmission. It then covers how to create, access, and update consecutive data sets for each type of transmission.

In a data set with consecutive organization, records are organized solely on the basis of their successive physical positions; when the data set is created, records are written consecutively in the order in which they are presented. You can retrieve the records only in the order in which they were written. See Table 10 on page 108 for valid file attributes and ENVIRONMENT options for consecutive data sets.

Using stream-oriented data transmission

This section covers how to define data sets for use with PL/I files that have the STREAM attribute. It covers the ENVIRONMENT options you can use and how to create and access data sets. The essential parameters of the DD statements you use in creating and accessing these data sets are summarized in tables, and several examples of PL/I programs are included to illustrate the text.

Data sets with the STREAM attribute are processed by stream-oriented data transmission, which allows your PL/I program to ignore block and record boundaries and treat a data set as a continuous stream of data values in character or graphic form.

You create and access data sets for stream-oriented data transmission using the list-, data-, and edit-directed input and output statements described in the *PL/I Language Reference*.

For output, PL/I converts the data items from program variables into character form if necessary, and builds the stream of characters or graphics into records for transmission to the data set.

For input, PL/I takes records from the data set and separates them into the data items requested by your program, converting them into the appropriate form for assignment to program variables.

You can use stream-oriented data transmission to read or write graphic data. There are terminals, printers, and data-entry devices that, with the appropriate programming support, can display, print, and enter graphics. You must be sure that your data is in a format acceptable for the intended device, or for a print utility program.

Defining files using stream I/O

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

```
DCL filename FILE STREAM
             INPUT | {OUTPUT [PRINT]}
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 10 on page 108; the FILE attribute is described in the *PL/I Language Reference*. The PRINT attribute is described further in “Using PRINT files with stream I/O” on page 130. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

Table 10 on page 108 summarizes the ENVIRONMENT options. The options applicable to stream-oriented data transmission are:

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
F|FB|V|VB|U
RECSIZE(record-length)
BLKSIZE(block-size)
GRAPHIC
```

BLKSIZE is described in Chapter 5, “Using data sets and files,” beginning on page 110. Descriptions of the rest of these options follow immediately below.

CONSECUTIVE

STREAM files must have CONSECUTIVE data set organization; however, it is not necessary to specify this in the ENVIRONMENT options since CONSECUTIVE is the default data set organization. The CONSECUTIVE option for STREAM files is the same as that described in “Data set organization” on page 102.

```
►►──CONSECUTIVE──────────────────────────────────────────────────►◄
```

Record format options

Although record boundaries are ignored in stream-oriented data transmission, record format is important when creating a data set. This is not only because record format affects the amount of storage space occupied by the data set and the efficiency of the program that processes the data, but also because the data set can later be processed by record-oriented data transmission.

Having specified the record format, you need not concern yourself with records and blocks as long as you use stream-oriented data transmission. You can consider your data set a series of characters or graphics arranged in lines, and you can use the SKIP option or format item (and, for a PRINT file, the PAGE and LINE options and format items) to select a new line.

```
►►──┬─F──┬───────────────────────────────────────────────────────►◄
    ├─FB─┤
    ├─V──┤
    ├─VB─┤
    └─U──┘
```

Records can have one of the following formats, which are described in “Record formats” on page 101.

Fixed-length	F	unblocked
	FB	blocked
Variable-length	V	unblocked
	VB	blocked
Undefined-length	U	(cannot be blocked)

Blocking and deblocking of records are performed automatically.

RECSIZE

RECSIZE for stream-oriented data transmission is the same as that described in “Specifying characteristics in the ENVIRONMENT attribute” on page 107. Additionally, a value specified by the LINESIZE option of the OPEN statement overrides a value specified in the RECSIZE option. LINESIZE is discussed in the *PL/I Language Reference*.

Additional record-size considerations for list- and data-directed transmission of graphics are given in the *PL/I Language Reference*.

Defaults for record format, BLKSIZE, and RECSIZE

If you do not specify the record format, BLKSIZE, or RECSIZE option in the ENVIRONMENT attribute, or in the associated DD statement or data set label, the following action is taken:

Input files:
Defaults are applied as for record-oriented data transmission, described in “Record format, BLKSIZE, and RECSIZE defaults” on page 111.

Output files:

Record format
Set to VB-format

Record length
The specified or default LINESIZE value is used:

PRINT files:

`F, FB, or U:`	line size + 1
`V or VB:`	line size + 5

Non-PRINT files:

`F, FB, or U:`	linesize
`V or VB:`	linesize + 4

Block size:

`F or FB:`	record length
`V or VB:`	record length + 4

GRAPHIC option

Specify the GRAPHIC option for edit-directed I/O.

```
►►──GRAPHIC──────────────────────────────────────────────────────►◄
```

The ERROR condition is raised for list- and data-directed I/O if you have graphics in input or output data and do not specify the GRAPHIC option.

For edit-directed I/O, the GRAPHIC option specifies that left and right delimiters are added to DBCS variables and constants on output, and that input graphics will have left and right delimiters. If you do not specify the GRAPHIC option, left and right delimiters are not added to output data, and input graphics do not require left and right delimiters. When you do specify the GRAPHIC option, the ERROR condition is raised if left and right delimiters are missing from the input data.

For information on the graphic data type, and on the G-format item for edit-directed I/O, see the *PL/I Language Reference*.

Creating a data set with stream I/O

To create a data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. For OS/390 UNIX, use one of the following to provide the additional information:

- TITLE option of the OPEN statement
- DD_DDNAME environment variable
- ENVIRONMENT attribute

The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

Essential information

When your application creates a STREAM file, it must supply a line-size value for that file from one of the following sources:

- LINESIZE option of the OPEN statement

  If you choose the LINESIZE option, it overrides all other sources.
- RECSIZE option of the ENVIRONMENT attribute

  The RECSIZE option of the ENVIRONMENT attribute overrides the other RECSIZE options.
- RECSIZE option of the TITLE option of the OPEN statement

  RECSIZE specified in the TITLE option of the OPEN statement has precedence over the RECSIZE option of the DD_DDNAME environment variable.
- RECSIZE option of the DD_DDNAME environment variable
- PL/I-supplied default value

  the PL/I default is used when you do not supply any value.

If LINESIZE is not supplied, but a RECSIZE value is, PL/I derives the line-size value from RECSIZE as follows:

- 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 a RECSIZE value of 1
- In all other cases, the value of RECSIZE is assigned to the line-size value.

PL/I determines a default line-size value based on attributes of the file and the type of associated data set. In cases where PL/I cannot supply an appropriate default line size, the UNDEFINEDFILE condition is raised.

A default line-size value is supplied for an OUTPUT file when:

- The file has the PRINT attribute. In this case, the value is obtained from the tab control table.
- The associated data set is the terminal (stdout: or stderr:). In this case the value is 120.

PL/I always derives the record length of the data set from the line-size value. A record-length value is derived from the line-size value as follows:

- For a 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
- In all other cases, the line-size value is assigned to the record-length value

Examples

The use of edit-directed stream-oriented data transmission to create a data set on a direct access storage device is shown in Figure 23. The data read from the input stream by the file SYSIN includes a field VREC that contains five unnamed 7-character subfields; the field NUM defines the number of these subfields that contain information. The output file WORK transmits to the data set the whole of the field FREC and only those subfields of VREC that contain information.

```
//EX7#2  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 PEOPLE: PROC OPTIONS(MAIN);
        DCL WORK FILE STREAM OUTPUT,
            1 REC,
              2 FREC,
                3 NAME CHAR(19),
                3 NUM CHAR(1),
                3 PAD CHAR(25),
              2 VREC CHAR(35),
            EOF BIT(1) INIT('0'B),
            IN CHAR(80) DEF REC;
        ON ENDFILE(SYSIN) EOF='1'B;
        OPEN FILE(WORK) LINESIZE(400);
        GET FILE(SYSIN) EDIT(IN)(A(80));
        DO WHILE (¬EOF);
        PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
        GET FILE(SYSIN) EDIT(IN)(A(80));
        END;
        CLOSE FILE(WORK);
        END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(NEW,CATLG),UNIT=SYSDA,
//           SPACE=(TRK,(1,1))
//GO.SYSIN DD *
R.C.ANDERSON      0 202848 DOCTOR
B.F.BENNETT       2 771239 PLUMBER          VICTOR HAZEL
R.E.COLE          5 698635 COOK             ELLEN  VICTOR JOAN   ANN    OTTO
J.F.COOPER        5 418915 LAWYER           FRANK  CAROL  DONALD NORMAN BRENDA
A.J.CORNELL       3 237837 BARBER           ALBERT ERIC   JANET
E.F.FERRIS        4 158636 CARPENTER        GERALD ANNA   MARY   HAROLD
/*
```

Figure 23. Creating a data set with stream-oriented data transmission

Figure 24 on page 128 shows an example of a program using list-directed output to write graphics to a stream file. It assumes that you have an output device that can print graphic data. The program reads employee records and selects persons living in a certain area. It then edits the address field, inserting one graphic blank between each address item, and prints the employee number, name, and address.

```
//EX7#3  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN      DD *
% PROCESS GRAPHIC;
  XAMPLE1:  PROC OPTIONS(MAIN);
            DCL  INFILE FILE INPUT RECORD,
                 OUTFILE FILE OUTPUT STREAM ENV(GRAPHIC);
 /* GRAPHIC OPTION MEANS DELIMITERS WILL BE INSERTED ON OUTPUT FILES. */
           DCL
                1 IN,
                   3 EMPNO CHAR(6),
                   3 SHIFT1 CHAR(1),
                   3 NAME,
                      5 LAST G(7),
                      5 FIRST G(7),
                   3 SHIFT2 CHAR(1),
                   3 ADDRESS,
                      5 ZIP CHAR(6),
                      5 SHIFT3 CHAR(1),
                      5 DISTRICT G(5),
                      5 CITY G(5),
                      5 OTHER G(8),
                      5 SHIFT4 CHAR(1);
           DCL EOF BIT(1) INIT('0'B);
           DCL ADDRWK G(20);
     ON ENDFILE (INFILE) EOF = '1'B;
     READ FILE(INFILE) INTO(IN);
     DO WHILE(¬EOF);
             DO;
                 IF SUBSTR(ZIP,1,3)¬='300'
                    THEN LEAVE;
                 L=0;
                 ADDRWK=DISTRICT;
                 DO I=1 TO 5;
                 IF SUBSTR(DISTRICT,I,1)= <'     ' G>
                    THEN LEAVE;              /* SUBSTR BIF PICKS 3P  */
                 END;                        /* THE ITH GRAPHIC CHAR */
                 L=L+I+1;                    /* IN DISTRICT          */
                 SUBSTR(ADDRWK,L,5)=CITY;
                 DO I=1 TO 5;
                 IF SUBSTR(CITY,I,1)= <'     ' G>
                    THEN LEAVE;
                 END;
                 L=L+I;
                 SUBSTR(ADDRWK,L,8)=OTHER;
                 PUT FILE(OUTFILE) SKIP          /* THIS DATA SET    */
                 EDIT(EMPNO,IN.LAST,FIRST,ADDRWK) /* REQUIRES UTILITY */
                     (A(8),G(7),G(7),X(4),G(20)); /* TO PRINT GRAPHIC */
                                                  /* DATA             */
                 END;                     /* END OF NON-ITERATIVE DO */
         READ FILE(INFILE) INTO (IN);
         END;                              /* END OF DO WHILE(¬EOF) */
     END XAMPLE1;
/*
//GO.OUTFILE   DD  SYSOUT=A,DCB=(RECFM=VB,LRECL=121,BLKSIZE=129)
//GO.INFILE DD *
ABCDEF< 山山山山山山山日日日日日日日>300099<          3 3 3 3 3 3 3   >
ABCD  < 山山山山      日日日日      >300011<          3 3 3 3       >
/*
```

Figure 24. Writing graphic data to a stream file

Accessing a data set with stream I/O

A data set accessed using stream-oriented data transmission need not have been created by stream-oriented data transmission, but it must have CONSECUTIVE organization, and all the data in it must be in character or graphic form. You can open the associated file for input, and read the records the data set contains; or you can open the file for output, and extend the data set by adding records at the end.

To access a data set, you must use one of the following to identify it:

- ENVIRONMENT attribute
- DD_DDNAME environment variable
- TITLE option of the OPEN statement

The following paragraphs describe the essential information you must include in the DD statement, and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

Essential information

When your application accesses an existing STREAM file, PL/I must obtain a record-length value for that file. The value can come from one of the following sources:

- The LINESIZE option of the OPEN statement
- The RECSIZE option of the ENVIRONMENT attribute
- The RECSIZE option of the DD_DDNAME environment variable
- The RECSIZE option of the TITLE option of the OPEN statement
- PL/I-supplied default value

If you are using an existing OUTPUT file, or if you supply a RECSIZE value, PL/I determines the record-length value as described in “Creating a data set with stream I/O” on page 126.

PL/I uses a default record-length value for an INPUT file when:

- The file is SYSIN, value = 80
- The file is associated with the terminal (stdout: or stderr:), value = 120

Record format

When using stream-oriented data transmission to access a data set, you do not need to know the record format of the data set (except when you must specify a block size); each GET statement transfers a discrete number of characters or graphics to your program from the data stream.

If you do give record-format information, it must be compatible with the actual structure of the data set. For example, if a data set is created with F-format records, a record size of 600 bytes, and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes; but if you specify a block size of 3500 bytes, your data will be truncated.

Example

The program in Figure 25 reads the data set created by the program in Figure 23 on page 127 and uses the file SYSPRINT to list the data it contains. (For details on SYSPRINT, see “Using SYSIN and SYSPRINT files” on page 134.) Each set of data is read, by the GET statement, into two variables: FREC, which always contains 45 characters; and VREC, which always contains 35 characters. At each execution of the GET statement, VREC consists of the number of characters generated by the expression 7*NUM, together with sufficient blanks to bring the total number of characters to 35. The DISP parameter of the DD statement could read simply DISP=OLD; if DELETE is omitted, an existing data set will not be deleted.

```
//EX7#5  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM INPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 SERNO CHAR(7),
                 3 PROF CHAR(18),
               2 VREC CHAR(35),
             IN CHAR(80) DEF REC,
             EOF BIT(1) INIT('0'B);
         ON ENDFILE(WORK) EOF='1'B;
         OPEN FILE(WORK);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
         GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         END;
         CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)
```

Figure 25. Accessing a data set with stream-oriented data transmission

Using PRINT files with stream I/O

Both the operating system and the PL/I language include features that facilitate the formatting of printed output. The operating system allows you to use the first byte of each record for a print control character. The control characters, which are not printed, cause the printer to skip to a new line or page. (Tables of print control characters are given in Figure 28 on page 141 and Figure 29 on page 141.)

In a PL/I program, the use of a PRINT file provides a convenient means of controlling the layout of printed output from stream-oriented data transmission. The compiler automatically inserts print control characters in response to the PAGE, SKIP, and LINE options and format items.

You can apply the PRINT attribute to any STREAM OUTPUT file, even if you do not intend to print the associated data set directly. When a PRINT file is associated with a direct-access data set, the print control characters have no effect on the layout of the data set, but appear as part of the data in the records.

The compiler reserves the first byte of each record transmitted by a PRINT file for an American National Standard print control character, and inserts the appropriate characters automatically.

A PRINT file uses only the following five print control characters:

Character	Action
	Space 1 line before printing (blank character)
0	Space 2 lines before printing
–	Space 3 lines before printing
+	No space before printing
1	Start new page

The compiler handles the PAGE, SKIP, and LINE options or format items by padding the remainder of the current record with blanks and inserting the appropriate control character in the next record. If SKIP or LINE specifies more than a 3-line space, the compiler inserts sufficient blank records with appropriate control characters to accomplish the required spacing. In the absence of a print control option or format item, when a record is full the compiler inserts a blank character (single line space) in the first byte of the next record.

If a PRINT file is being transmitted to a terminal, the PAGE, SKIP, and LINE options will never cause more than 3 lines to be skipped, unless formatted output is specified.

Controlling printed line length

You can limit the length of the printed line produced by a PRINT file either by specifying a record length in your PL/I program (ENVIRONMENT attribute) or in a DD statement, or by giving a line size in an OPEN statement (LINESIZE option). The record length must include the extra byte for the print control character, that is, it must be 1 byte larger than the length of the printed line (5 bytes larger for V-format records). The value you specify in the LINESIZE option refers to the number of characters in the printed line; the compiler adds the print control character.

The blocking of records has no effect on the appearance of the output produced by a PRINT file, but it does result in more efficient use of auxiliary storage when the file is associated with a data set on a direct-access device. If you use the LINESIZE option, ensure that your line size is compatible with your block size. For F-format records, block size must be an exact multiple of (line size+1); for V-format records, block size must be at least 9 bytes greater than line size.

Although you can vary the line size for a PRINT file during execution by closing the file and opening it again with a new line size, you must do so with caution if you are using the PRINT file to create a data set on a direct-access device. You cannot change the record format that is established for the data set when the file is first opened. If the line size you specify in an OPEN statement conflicts with the record format already established, the UNDEFINEDFILE condition is raised. To prevent this, either specify V-format records with a block size at least 9 bytes greater than the maximum line size you intend to use, or ensure that the first OPEN statement specifies the maximum line size. (Output destined for the printer can be stored temporarily on a direct-access device, unless you specify a printer by using UNIT=, even if you intend it to be fed directly to the printer.)

Since PRINT files have a default line size of 120 characters, you need not give any record format information for them. In the absence of other information, the compiler assumes V-format records. The complete default information is:

BLKSIZE=129

LRECL=125

RECFM=VBA.

Example: Figure 26 on page 133 illustrates the use of a PRINT file and the printing options of stream-oriented data transmission statements to format a table and write it onto a direct-access device for printing on a later occasion. The table comprises the natural sines of the angles from 0° to 359° 54' in steps of 6'.

The statements in the ENDPAGE ON-unit insert a page number at the bottom of each page, and set up the headings for the following page.

The DD statement defining the data set created by this program includes no record-format information. The compiler infers the following from the file declaration and the line size specified in the statement that opens the file TABLE:

Record format =	V (the default for a PRINT file).
Record size =	98 (line size + 1 byte for print control character + 4 bytes for record control field).
Block size =	102 (record length + 4 bytes for block control field).

The program in Figure 31 on page 146 uses record-oriented data transmission to print the table created by the program in Figure 26 on page 133.

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

 SINE: PROC OPTIONS(MAIN);
   DCL TABLE       FILE STREAM OUTPUT PRINT;
   DCL DEG         FIXED DEC(5,1) INIT(0);  /* INIT(0) FOR ENDPAGE  */
   DCL MIN         FIXED DEC(3,1);
   DCL PGNO        FIXED DEC(2)   INIT(0);
   DCL ONCODE      BUILTIN;

   ON ERROR
     BEGIN;
       ON ERROR SYSTEM;
       DISPLAY ('ONCODE = '|| ONCODE);
     END;

   ON ENDPAGE(TABLE)
     BEGIN;
       DCL I;
       IF PGNO ¬= 0 THEN
         PUT FILE(TABLE) EDIT ('PAGE',PGNO)
             (LINE(55),COL(80),A,F(3));
       IF DEG ¬= 360 THEN
         DO;
           PUT FILE(TABLE) PAGE EDIT ('NATURAL SINES') (A);
           IF PGNO ¬= 0 THEN
             PUT FILE(TABLE) EDIT ((I DO I = 0 TO 54 BY 6))
                                  (SKIP(3),10 F(9));
           PGNO = PGNO + 1;
         END;
       ELSE
         PUT FILE(TABLE) PAGE;
     END;

   OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
   SIGNAL ENDPAGE(TABLE);

   PUT FILE(TABLE) EDIT
    ((DEG,(SIND(DEG+MIN) DO MIN = 0 TO .9 BY .1) DO DEG = 0 TO 359))
    (SKIP(2), 5 (COL(1), F(3), 10 F(9,4) ));
   PUT FILE(TABLE) SKIP(52);
 END SINE;
```

Figure 26. Creating a print file via stream data transmission. The example in Figure 31 on page 146 will print the resultant file.

Overriding the tab control table

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions. See Figure 14 on page 72 and Figure 27 on page 134 for examples of declaring a tab table. The definitions of the fields in the table are as follows:

OFFSET OF TAB COUNT:
: Halfword binary integer that gives the offset of "Tab count," the field that indicates the number of tabs to be used.

PAGESIZE:
: Halfword binary integer that defines the default page size. This page size is used for dump output to the PLIDUMP data set as well as for stream output.

LINESIZE: Halfword binary integer that defines the default line size.

PAGELENGTH:
: Halfword binary integer that defines the default page length for printing at a terminal.

FILLERS: Three halfword binary integers; reserved for future use.

TAB COUNT:
: Halfword binary integer that defines the number of tab position entries in the table (maximum 255). If tab count = 0, any specified tab positions are ignored.

Tab1–Tab*n*:
: *n* halfword binary integers that define the tab positions within the print line. The first position is numbered 1, and the highest position is numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it is ignored. The first data field in the printed output begins at the next available tab position.

You can override the default PL/I tab settings for your program by causing the linkage editor to resolve an external reference to PLITABS. To cause the reference to be resolved, supply a table with the name PLITABS, in the format described above.

To supply this tab table, include a PL/I structure in your source program with the name PLITABS, which you must declare to be STATIC EXTERNAL in your MAIN proc. An example of the PL/I structure is shown in Figure 27. 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 27. PL/I structure PLITABS for modifying the preset tab settings

Using SYSIN and SYSPRINT files

If you code a GET statement without the FILE option in your program, the compiler inserts the file name SYSIN. If you code a PUT statement without the FILE option, the compiler inserts the name SYSPRINT.

If you do not declare SYSPRINT, the compiler gives the file the attribute PRINT in addition to the normal default attributes; the complete set of attributes will be:

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

Since SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 characters and a V-format record. You need give only a minimum of information in the corresponding DD statement; if your installation uses the usual convention that the system output device of class A is a printer, the following is sufficient:

```
//SYSPRINT DD SYSOUT=A
```

Note: SYSIN and SYSPRINT are established in the User Exit during initialization. IBM-supplied defaults for SYSIN and SYSPRINT are directed to the terminal.

You can override the attributes given to SYSPRINT by the compiler by explicitly declaring or opening the file. For more information about the interaction between SYSPRINT and the Language Environment for OS/390 & VM message file option, see the *OS/390 Language Environment Programming Guide*.

The compiler does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes. The data set associated with SYSIN is usually in the input stream; if it is not in the input stream, you must supply full DD information.

For more information about SYSPRINT, see "SYSPRINT considerations" on page 74.

Controlling input from the terminal

You can enter data at the terminal for an input file in your PL/I program if you do the following:

1. Declare the input file explicitly or implicitly with the CONSECUTIVE environment option (all stream files meet this condition)
2. Allocate the input file to the terminal.

You can usually use the standard default input file SYSIN because it is a stream file and can be allocated to the terminal.

You are prompted for input to stream files by a colon (:). You will see the colon each time a GET statement is executed in the program. The GET statement causes the system to go to the next line. You can then enter the required data. If you enter a line that does not contain enough data to complete execution of the GET statement, a further prompt, which is a plus sign followed by a colon (+:), is displayed.

By adding a hyphen to the end of any line that is to continue, you can delay transmission of the data to your program until you enter two or more lines.

If you include output statements that prompt you for input in your program, you can inhibit the initial system prompt by ending your own prompt with a colon. For example, the GET statement could be preceded by a PUT statement such as:

```
PUT SKIP LIST('ENTER NEXT ITEM:');
```

To inhibit the system prompt for the next GET statement, your own prompt must meet the following conditions:

1. It must be either list-directed or edit-directed, and if list-directed, must be to a PRINT file.
2. The file transmitting the prompt must be allocated to the terminal. If you are merely copying the file at the terminal, the system prompt is not inhibited.

Format of data

The data you enter at the terminal should have exactly the same format as stream input data in batch mode, except for the following variations:

- Simplified punctuation for input: If you enter separate items of input on separate lines, there is no need to enter intervening blanks or commas; the compiler will insert a comma at the end of each line.

  For instance, in response to the statement:

  ```
  GET LIST(I,J,K);
  ```

  your terminal interaction could be as follows:

  ```
  :
  1
  +:2
  +:3
  ```

  with a carriage return following each item. It would be equivalent to:

  ```
  :
  1,2,3
  ```

  If you wish to continue an item onto another line, you must end the first line with a continuation character. Otherwise, for a GET LIST or GET DATA statement, a comma will be inserted, and for a GET EDIT statement, the item will be padded (see next paragraph).

- Automatic padding for GET EDIT: There is no need to enter blanks at the end of a line of input for a GET EDIT statement. The item you enter will be padded to the correct length.

  For instance, for the PL/I statement:

  ```
  GET EDIT(NAME)(A(15));
  ```

  you could enter the five characters:

  ```
  SMITH
  ```

  followed immediately by a carriage return. The item will be padded with 10 blanks, so that the program receives a string 15 characters long. If you wish to continue an item on a second or subsequent line, you must add a continuation character to the end of every line except the last; the first line transmitted would otherwise be padded and treated as the complete data item.

- SKIP option or format item: A SKIP in a GET statement asks the program to ignore data not yet entered. All uses of SKIP(n) where `n` is greater than one are taken to mean SKIP(1). SKIP(1) is taken to mean that all unused data on the current line is ignored.

Stream and record files

You can allocate both stream and record files to the terminal. However, no prompting is provided for record files. If you allocate more than one file to the terminal, and one or more of them is a record file, the output of the files will not necessarily be synchronized. The order in which data is transmitted to and from the terminal is not guaranteed to be the same order in which the corresponding PL/I I/O statements are executed.

Also, record file input from the terminal is received in upper case letters because of a TCAM restriction. To avoid problems you should use stream files wherever possible.

Capital and lowercase letters

For stream files, character strings are transmitted to the program as entered in lowercase or uppercase. For record files, all characters become uppercase.

End-of-file

The characters /* in positions one and two of a line that contains no other characters are treated as an end-of-file mark, that is, they raise the ENDFILE condition.

COPY option of GET statement

The GET statement can specify the COPY option; but if the COPY file, as well as the input file, is allocated to the terminal, no copy of the data will be printed.

Controlling output to the terminal

At your terminal you can obtain data from a PL/I file that has been both:

1. Declared explicitly or implicitly with the CONSECUTIVE environment option. All stream files meet this condition.
2. Allocated to the terminal.

The standard print file SYSPRINT generally meets both these conditions.

Format of PRINT files

Data from SYSPRINT or other PRINT files is not normally formatted into pages at the terminal. Three lines are always skipped for PAGE and LINE options and format items. The ENDPAGE condition is normally never raised. SKIP(n), where n is greater than three, causes only three lines to be skipped. SKIP(0) is implemented by backspacing, and should therefore not be used with terminals that do not have a backspace feature.

You can cause a PRINT file to be formatted into pages by inserting a tab control table in your program. The table must be called PLITABS, and its contents are explained in “Overriding the tab control table” on page 133. You must initialize the element PAGELENGTH to the length of page you require—that is, the length of the sheet of paper on which each page is to be printed, expressed as the maximum number of lines that could be printed on it. You must initialize the element PAGESIZE to the actual number of lines to be printed on each page. After the number of lines in PAGESIZE has been printed on a page, ENDPAGE is raised, for which standard system action is to skip the number of lines equal to PAGELENGTH minus PAGESIZE, and then start printing the next page. For other than standard layout, you must initialize the other elements in PLITABS to the values shown in Figure 14 on page 72. You can also use PLITABS to alter the tabulating positions of list-directed and data-directed output. You can use PLITABS for SYSPRINT when you need to format page breaks in ILC applications. Set PAGESIZE to 32767 and use the PUT PAGE statement to control page breaks.

Although some types of terminals have a tabulating facility, tabulating of list-directed and data-directed output is always achieved by transmission of blank characters.

Stream and record files

You can allocate both stream and record files to the terminal. However, if you allocate more than one file to the terminal and one or more is a record file, the files' output will not necessarily be synchronized. There is no guarantee that the order in which data is transmitted between the program and the terminal will be the same as the order in which the corresponding PL/I input and output statements are executed. In addition, because of a TCAM restriction, any output to record files at the terminal is printed in uppercase (capital) letters. It is therefore advisable to use stream files wherever possible.

Capital and lowercase characters

For stream files, characters are displayed at the terminal as they are held in the program, provided the terminal can display them. For instance, with an IBM 327x terminal, capital and lowercase letters are displayed as such, without translation. For record files, all characters are translated to uppercase. A variable or constant in the program can contain lowercase letters if the program was created under the EDIT command with the ASIS operand, or if the program has read lowercase letters from the terminal.

Output from the PUT EDIT command

The format of the output from a PUT EDIT command to a terminal is line mode TPUTs with “Start of field” and “end of field” characters appearing as blanks on the screen.

Using record-oriented data transmission

PL/I supports various types of data sets with the RECORD attribute (see Table 14 on page 142). This section covers how to use consecutive data sets.

Table 13 lists the statements and options that you can use to create and access a consecutive data set using record-oriented data transmission.

Table 13 (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	WRITE FILE(file-reference) FROM(reference);	

Table 13 (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 INPUT BUFFERED	READ FILE(file-reference) INTO(reference);
	
	READ FILE(file-reference) SET(pointer-reference);
	
	READ FILE(file-reference) IGNORE(expression);	

SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);
	
	READ FILE(file-reference) IGNORE(expression);	

SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);
	
	READ FILE(file-reference) SET(pointer-reference);
	
	READ FILE(file-reference) IGNORE(expression);
	
	REWRITE FILE(file-reference);
		
		FROM(reference)

SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference);
	
	READ FILE(file-reference) IGNORE(expression);
	
	REWRITE FILE(file-reference) FROM(reference);	

Notes:

1. The complete file declaration would include the attributes FILE, RECORD and ENVIRONMENT.
2. The statement READ FILE (file-reference); is a valid statement and is equivalent to READ FILE(file-reference) IGNORE (1);

Specifying record format

If you give record-format information, it must be compatible with the actual structure of the data set. For example, if you create a data set with FB-format records, with a record size of 600 bytes and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes. If you specify a block size of 3500 bytes, your data is truncated.

Defining files using record I/O

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 10 on page 108. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to consecutive data sets are:

```
F|FB|V|VB|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING

CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
CTLASA|CTL360
```

The options above the blank line are described in “Specifying characteristics in the ENVIRONMENT attribute” on page 107, and those below the blank line are described below.

See Table 10 on page 108 to find which options you must specify, which are optional, and which are defaults.

CONSECUTIVE

The CONSECUTIVE option defines a file with consecutive data set organization, which is described in this chapter and in “Data set organization” on page 102.

```
►►──CONSECUTIVE─────────────────────────────────────────────────────►◄
```

CONSECUTIVE is the default.

ORGANIZATION(CONSECUTIVE)

Specifies that the file is associated with a consecutive data set. The ORGANIZATION option is described in “ORGANIZATION option” on page 114.

The file can be either a native data set or a VSAM data set.

CTLASA|CTL360

The printer control options CTLASA and CTL360 apply only to OUTPUT files associated with consecutive data sets. They specify that the first character of a record is to be interpreted as a control character.

```
►►──┬─CTLASA─┬──────────────────────────────────────────────────────►◄
    └─CTL360─┘
```

The CTLASA option specifies American National Standard Vertical Carriage Positioning Characters or American National Standard Pocket Select Characters (Level 1). The CTL360 option specifies IBM machine-code control characters.

The American National Standard control characters, listed in Figure 28 on page 141, cause the specified action to occur before the associated record is printed or punched.

The machine code control characters differ according to the type of device. The IBM machine code control characters for printers are listed in Figure 29 on page 141.

Code	Action
	Space 1 line before printing (blank code)
0	Space 2 lines before printing
−	Space 3 lines before printing
+	Suppress space before printing
1	Skip to channel 1
2	Skip to channel 2
3	Skip to channel 3
4	Skip to channel 4
5	Skip to channel 5
6	Skip to channel 6
7	Skip to channel 7
8	Skip to channel 8
9	Skip to channel 9
A	Skip to channel 10
B	Skip to channel 11
C	Skip to channel 12
V	Select stacker 1
W	Select stacker 2

Figure 28. American National Standard print and card punch control characters (CTLASA)

Print and Then Act	Action	Act immediately (no printing)
Code byte		**Code byte**
00000001	Print only (no space)	—
00001001	Space 1 line	00001011
00010001	Space 2 lines	00010011
00011001	Space 3 lines	00011011
10001001	Skip to channel 1	10001011
10010001	Skip to channel 2	10010011
10011001	Skip to channel 3	10011011
10100001	Skip to channel 4	10100011
10101001	Skip to channel 5	10101011
10110001	Skip to channel 6	10110011
10111001	Skip to channel 7	10111011
11000001	Skip to channel 8	11000011
11001001	Skip to channel 9	11001011
11010001	Skip to channel 10	11010011
11011001	Skip to channel 11	11011011
11100001	Skip to channel 12	11100011

Figure 29. IBM machine code print control characters (CTL360)

Creating a data set with record I/O

When you create a consecutive data set, you must open the associated file for SEQUENTIAL OUTPUT. You can use either the WRITE or LOCATE statement to write records. Table 13 on page 138 shows the statements and options for creating a consecutive data set.

When creating a data set, you must identify it to the operating system in a DD statement. The following paragraphs, summarized in Table 14 on page 142, tell what essential information you must include in the DD statement and discuss some of the optional information you can supply.

Table 14. Creating a consecutive data set with record I/O: essential parameters of the DD statement

Storage device	When required	What you must state	Parameters
All	Always	Output device	UNIT= or SYSOUT= or VOLUME=REF=
		Block size[1]	DCB=(BLKSIZE=...
Direct access only	Always	Storage space required	SPACE=
Direct access	Data set to be used by another job step but not required at end of job	Disposition	DISP=
	Data set to be kept after end of job	Disposition	DISP=
		Name of data set	DSNAME=
	Data set to be on particular device	Volume serial number	VOLUME=SER= or VOLUME=REF=

[1]Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

Essential information

When you create a consecutive data set you must specify:

- The name of data set to be associated with your PL/I file. A data set with consecutive organization can exist on any type of device.
- The record length. You can specify the record length using the RECSIZE option of the ENVIRONMENT attribute, of the DD_DDNAME environment variable, or of the TITLE option of the OPEN statement.

  For files associated with the terminal device (stdout: or stderr:), PL/I uses a default record length of 120 when the RECSIZE option is not specified.

Accessing and updating a data set with record I/O

Once you create a consecutive data set, you can open the file that accesses it for sequential input, for sequential output, or, for data sets on direct-access devices, for updating. See Figure 30 on page 144 for an example of a program that accesses and updates a consecutive data set. If you open the file for output, and extend the data set by adding records at the end, you must specify DISP=MOD in the DD statement. If you do not, the data set will be overwritten. If you open a file for updating, you can only update records in their existing sequence, and if you want to insert records, you must create a new data set. Table 13 on page 138 shows the statements and options for accessing and updating a consecutive data set.

When you access a consecutive data set by a SEQUENTIAL UPDATE file, you must retrieve a record with a READ statement before you can update it with a REWRITE statement; however, every record that is retrieved need not be rewritten. A REWRITE statement will always update the last record read.

Consider the following:

```
READ FILE(F) INTO(A);
     .
     .
     .
READ FILE(F) INTO(B);
     .
     .
     .
REWRITE FILE(F) FROM(A);
```

The REWRITE statement updates the record that was read by the second READ statement. The record that was read by the first statement cannot be rewritten after the second READ statement has been executed.

To access a data set, you must identify it to the operating system in a DD statement. Table 15 summarizes the DD statement parameters needed to access a consecutive data set.

Table 15. Accessing a consecutive data set with record I/O: essential parameters of the DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	
UNIT= or VOLUME=REF=	Input device	If data set not cataloged (all devices)
VOLUME=SER=	Volume serial number	If data set not cataloged (direct access)
DCB=(BLKSIZE=	Block size[1]	If data set does not have standard labels

[1]Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

The following paragraphs indicate the essential information you must include in the DD statement, and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

Essential information

If the data set is cataloged, you need to supply only the following information in the DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.
- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, opening the data set for output will result in it being overwritten.

If the data set is not cataloged, you must additionally specify the device that will read the data set and, direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

Example of consecutive data sets

Creating and accessing consecutive data sets are illustrated in the program in Figure 30. The program merges the contents of two data sets, in the input stream, and writes them onto a new data set, &&TEMP; each of the original data sets contains 15-byte fixed-length records arranged in EBCDIC collating sequence. The two input files, INPUT1 and INPUT2, have the default attribute BUFFERED, and locate mode is used to read records from the associated data sets into the respective buffers. Access of based variables in the buffers should not be attempted after the file has been closed.

```
//EXAMPLE  JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN   DD *
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 MERGE: PROC OPTIONS(MAIN);
   DCL (INPUT1,                            /* FIRST INPUT FILE     */
        INPUT2,                            /* SECOND INPUT FILE    */
        OUT )      FILE RECORD SEQUENTIAL; /* RESULTING MERGED FILE*/
   DCL SYSPRINT    FILE PRINT;             /* NORMAL PRINT FILE    */

   DCL INPUT1_EOF  BIT(1) INIT('0'B);      /* EOF FLAG FOR INPUT1  */
   DCL INPUT2_EOF  BIT(1) INIT('0'B);      /* EOF FLAG FOR INPUT2  */
   DCL OUT_EOF     BIT(1) INIT('0'B);      /* EOF FLAG FOR OUT     */
   DCL TRUE        BIT(1) INIT('1'B);      /* CONSTANT TRUE        */
   DCL FALSE       BIT(1) INIT('0'B);      /* CONSTANT FALSE       */

   DCL ITEM1       CHAR(15) BASED(A);      /* ITEM FROM INPUT1     */
   DCL ITEM2       CHAR(15) BASED(B);      /* ITEM FROM INPUT2     */
   DCL INPUT_LINE  CHAR(15);               /* INPUT FOR READ INTO  */
   DCL A           POINTER;                /* POINTER VAR          */
   DCL B           POINTER;                /* POINTER VAR          */

   ON ENDFILE(INPUT1) INPUT1_EOF = TRUE;
   ON ENDFILE(INPUT2) INPUT2_EOF = TRUE;
   ON ENDFILE(OUT)    OUT_EOF    = TRUE;

   OPEN FILE(INPUT1) INPUT,
        FILE(INPUT2) INPUT,
        FILE(OUT)    OUTPUT;

   READ FILE(INPUT1) SET(A);               /* PRIMING READ         */
   READ FILE(INPUT2) SET(B);

   DO WHILE ((INPUT1_EOF = FALSE) & (INPUT2_EOF = FALSE));
     IF ITEM1 > ITEM2 THEN
       DO;
         WRITE FILE(OUT) FROM(ITEM2);
         PUT FILE(SYSPRINT) SKIP EDIT('1>2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT2) SET(B);
       END;
     ELSE
       DO;
         WRITE FILE(OUT) FROM(ITEM1);
         PUT FILE(SYSPRINT) SKIP EDIT('1<2', ITEM1, ITEM2)
             (A(5),A,A);
         READ FILE(INPUT1) SET(A);
       END;
   END;
```

Figure 30 (Part 1 of 2). Merge Sort—creating and accessing a consecutive data set

```
   DO WHILE (INPUT1_EOF = FALSE);           /* INPUT2 IS EXHAUSTED  */
     WRITE FILE(OUT) FROM(ITEM1);
     PUT FILE(SYSPRINT) SKIP EDIT('1', ITEM1) (A(2),A);
     READ FILE(INPUT1) SET(A);
   END;

   DO WHILE (INPUT2_EOF = FALSE);           /* INPUT1 IS EXHAUSTED  */
     WRITE FILE(OUT) FROM(ITEM2);
     PUT FILE(SYSPRINT) SKIP EDIT('2', ITEM2) (A(2),A);
     READ FILE(INPUT2) SET(B);
   END;

   CLOSE FILE(INPUT1), FILE(INPUT2), FILE(OUT);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(OUT) SEQUENTIAL INPUT;

   READ FILE(OUT) INTO(INPUT_LINE);         /* DISPLAY OUT FILE     */
   DO WHILE (OUT_EOF = FALSE);
     PUT FILE(SYSPRINT) SKIP EDIT(INPUT_LINE) (A);
     READ FILE(OUT) INTO(INPUT_LINE);
   END;
   CLOSE FILE(OUT);

 END MERGE;
/*
//GO.INPUT1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.INPUT2 DD *
BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,
//         DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))
```

Figure 30 (Part 2 of 2). Merge Sort—creating and accessing a consecutive data set

The program in Figure 31 on page 146 uses record-oriented data transmission to print the table created by the program in Figure 26 on page 133.

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

 PRT: PROC OPTIONS(MAIN);
   DCL TABLE       FILE RECORD INPUT SEQUENTIAL;
   DCL PRINTER     FILE RECORD OUTPUT SEQL
                        ENV(V BLKSIZE(102) CTLASA);
   DCL LINE        CHAR(94) VAR;

   DCL TABLE_EOF   BIT(1) INIT('0'B);       /* EOF FLAG FOR TABLE   */
   DCL TRUE        BIT(1) INIT('1'B);       /* CONSTANT TRUE        */
   DCL FALSE       BIT(1) INIT('0'B);       /* CONSTANT FALSE       */


   ON ENDFILE(TABLE) TABLE_EOF = TRUE;

   OPEN FILE(TABLE),
        FILE(PRINTER);

   READ FILE(TABLE) INTO(LINE);             /* PRIMING READ         */

   DO WHILE (TABLE_EOF = FALSE);
     WRITE FILE(PRINTER) FROM(LINE);
     READ FILE(TABLE) INTO(LINE);
   END;

   CLOSE FILE(TABLE),
         FILE(PRINTER);
 END PRT;
```

Figure 31. Printing record-oriented data transmission

Chapter 8. Defining and using indexed data sets

This chapter describes data transmission statements and ENVIRONMENT options that define indexed data sets and how to create, access, and reorganize indexed data sets.

IMPORTANT: INDEXED currently implies VSAM and is supported only under batch.

Indexed organization

A data set with indexed organization must be on a direct-access device. Its records can be either F-format or V-format records, blocked or unblocked. The records are arranged in logical sequence, according to keys associated with each record. A *key* is a character string that can identify each record uniquely. Logical records are arranged in the data set in ascending key sequence according to the EBCDIC collating sequence. Indexes associated with the data set are used by the operating system data-management routines to locate a record when the key is supplied.

Unlike consecutive organization, indexed organization does not require you to access every record in sequential fashion. You must create an indexed data set sequentially; but once you create it, you can open the associated file for SEQUENTIAL or DIRECT access, as well as INPUT or UPDATE. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Sequential processing of an indexed data set is slower than that of a corresponding consecutive data set, because the records it contains are not necessarily retrieved in physical sequence. Furthermore, random access is less efficient for an indexed data set than for a regional data set, because the indexes must be searched to locate a record. An indexed data set requires more external storage space than a consecutive data set, and all volumes of a multivolume data set must be mounted, even for sequential processing.

Table 16 on page 148 lists the data-transmission statements and options that you can use to create and access an indexed data set.

Using keys

There are two kinds of keys—recorded keys and source keys. A *recorded key* is a character string that actually appears with each record in the data set to identify that record. The length of the recorded key cannot exceed 255 characters and all keys in a data set must have the same length. The recorded keys in an indexed data set can be separate from, or embedded within, the logical records. 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. For direct access of an indexed data set, you must include a source key in each transmission statement.

Note: All VSAM key-sequenced data sets have embedded keys.

Table 16 (Page 1 of 2). Statements and options allowed for creating and accessing indexed data sets

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

Table 16 (Page 2 of 2). Statements and options allowed for creating and accessing indexed data sets

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

DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression);
	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);
	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);
	
	DELETE FILE(file-reference) KEY(expression);[2]
	
	UNLOCK 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 in the file declaration. The attribute BUFFERED is the default for INDEXED SEQUENTIAL and SEQUENTIAL files.
2. Use of the DELETE statement is invalid if you did not specify OPTCD=L (DCB subparameter) when the data set was created.

The use of embedded keys avoids the need for the KEYTO option during sequential input, but the KEYFROM option is still required for output. (However, the data specified by the KEYFROM option can be the embedded key portion of the record variable itself.) In a data set with unblocked records, a separate recorded key precedes each record, even when there is already an embedded key. If the records are blocked, the key of only the last record in each block is recorded separately in front of the block.

During execution of a WRITE statement that adds a record to a data set with embedded keys, the value of the expression in the KEYFROM option is assigned to the embedded key position in the record variable. Note that you can declare a record variable as a structure with an embedded key declared as a structure member, but that you must not declare such an embedded key as a VARYING string.

For a REWRITE statement using SEQUENTIAL files with indexed data set organization, you must ensure that the rewritten key is the same as the key in the replaced record.

For a LOCATE statement, the KEYFROM string is assigned to the embedded key when the next operation on the file is encountered.

Using indexes

To provide faster access to the records in the data set, the operating system creates and maintains a system of indexes to the records in the data set.

The lowest level of index is the *track index*. There is a track index for each cylinder in the data set. The track index occupies the first track (or tracks) of the cylinder, and lists the key of the last record on each track in the cylinder. A search can then be directed to the first track that has a key that is higher than or equal to the key of the required record.

If the data set occupies more than one cylinder, the operating system develops a higher-level index called a *cylinder index*. Each entry in the cylinder index identifies the key of the last record in the cylinder.

To increase the speed of searching the cylinder index, you can request in a DD statement that the operating system develop a *master index* for a specified number of cylinders. You can have up to three levels of master index.

Figure 32 illustrates the index structure. The part of the data set that contains the cylinder and master indexes is termed the *index area*.

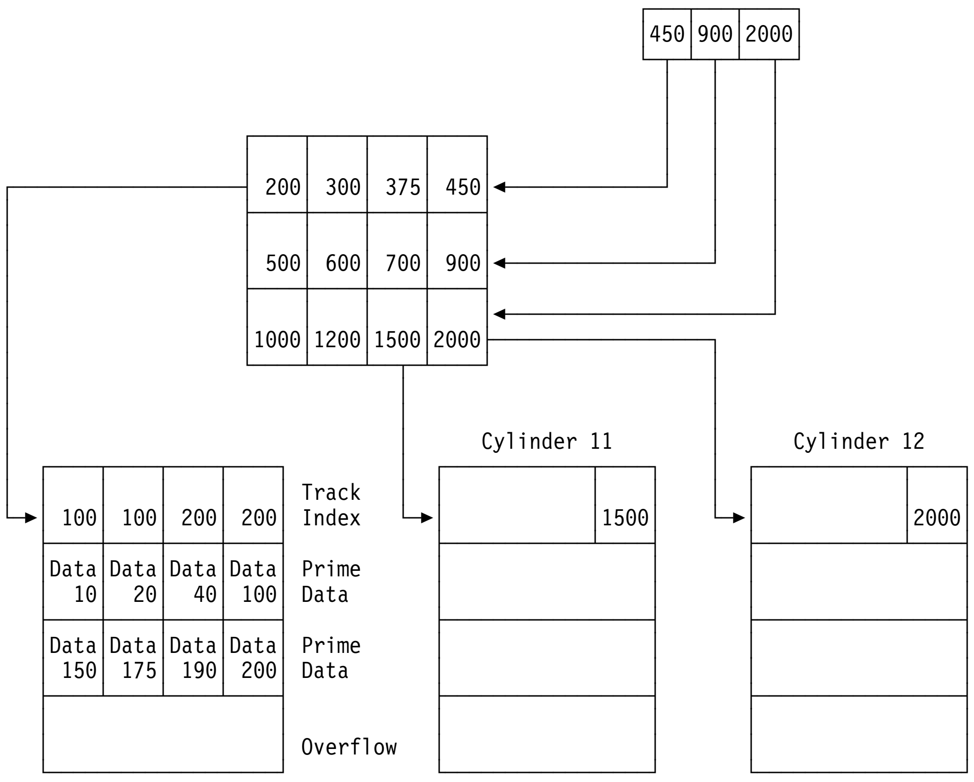


Figure 32. Index structure of an indexed data set

When you create an indexed data set, all the records are written in what is called the *prime data area*. If you add more records later, the operating system does not rearrange the entire data set; it inserts each new record in the appropriate position and moves up the other records on the same track. Any records forced off the track by the insertion of a new record are placed in an *overflow area*. The overflow area can be either a number of tracks set aside in each cylinder for the overflow records from that cylinder (*cylinder overflow area*), or a separate area for all overflow records (*independent overflow area*).

Records in the overflow area are chained together to the track index so as to maintain the logical sequence of the data set. This is illustrated in Figure 33 on page 152. Each entry in the track index consists of two parts:

- The normal entry, which points to the last record on the track
- The overflow entry, which contains the key of the first record transferred to the overflow area and also points to the last record transferred from the track to the overflow area.

If there are no overflow records from the track, both index entries point to the last record on the track. An additional field is added to each record that is placed in the overflow area. It points to the previous record transferred from the same track. The first record from each track is linked to the corresponding overflow entry in the track index.

Dummy records

Records within an indexed data set are either actual records, containing valid data, or dummy records. A dummy record, identified by the constant (8)'1'B in its first byte, can be one that you insert or it can be created by the operating system. You insert dummy records by setting the first byte to (8)'1'B and writing the records in the usual way. The operating system creates dummy records by placing (8)'1'B in a record that is named in a DELETE statement.

When creating an indexed data set, you might want to insert dummy records to reserve space in the prime data area. You can replace dummy records later with actual data records having the same key.

The operating system removes dummy records when the data set is reorganized, as described later in this section, and removes those forced off the track during an update.

If you include the DCB subparameter OPTCD=L in the DD statement that defines the data set when you create it, dummy records will not be retrieved by READ statements and the operating system will write the dummy identifier in records being deleted.

Defining files for an indexed data set

You define a sequential indexed data set by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

You define a direct indexed data set by a file declaration with the following attributes:

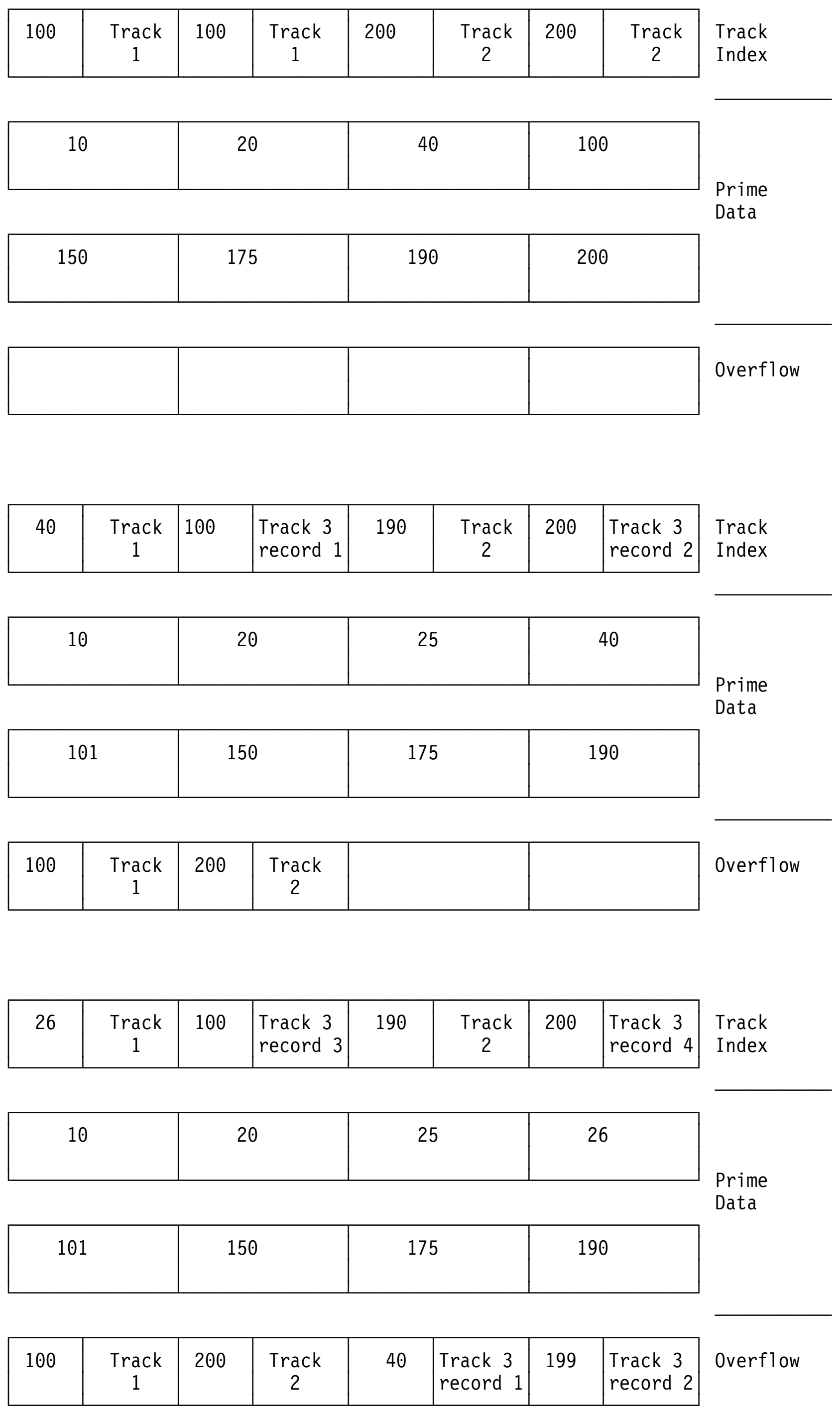


Figure 33. Adding records to an indexed data set

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             KEYED
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 10 on page 108. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to indexed data sets are:

```
F|FB|V|VB
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
KEYLENGTH(n)
GENKEY

KEYLOC(n)
```

The options above the blank line are described in “Specifying characteristics in the ENVIRONMENT attribute” on page 107, and those below the blank line are described below.

KEYLOC option — key location

Use the KEYLOC option with indexed data sets when you create the data set to specify the starting position of an embedded key in a record.

```
►►──KEYLOC──────────────────────────────────────────────────────────►◄
```

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.

If the keys are embedded within the records, specify the KEYLOC option.

The KEYLOC option specifies the absolute position of an embedded key from the start of the data in a record.

Thus the equivalent KEYLOC values for a particular byte are affected by the following:

- The KEYLOC byte count starts at 1; the RKP count starts at 0.
- The record format.

For example, if the embedded key begins at the tenth byte of a record variable, the specifications are:

Fixed-length:

```
KEYLOC(10)
RKP=9
```

Variable-length:

```
KEYLOC(10)
RKP=13
```

If KEYLOC is specified with a value equal to or greater than 1, embedded keys exist in the record variable and on the data set. If KEYLOC is equal to zero, or is not specified, the RKP value is used. When RKP is specified, the key is part of the variable only when RKP≥1. As a result, embedded keys might not always be present in the record variable or the data set. If you specify `KEYLOC(1)`, you must specify it for every file that accesses the data set. This is necessary because KEYLOC(1) cannot be converted to an unambiguous RKP value. (Its equivalent is RKP=0 for fixed format, which in turn implies nonembedded keys.) The effect of the use of both options is shown in Table 17.

Table 17. Effect of KEYLOC and RKP values on establishing embedded keys in record variables or data sets

KEYLOC(n)	**RKP**	**Record variable**	**Data set unblocked records**	**Data set blocked records**
n>1	RKP equivalent = n−1+C[1]	Key	Key	Key
n=1	No equivalent	Key	Key[2]	Key
n=0 or not specified	RKP=C[1]	No Key	No Key	Key[3]
	RKP>C[1]	Key	Key	Key

Notes:

1. C = number of control bytes, if any:
   C=0 for fixed-length records.
   C=4 for variable-length records.
2. In this instance the key is not recognized by data management .
3. Each logical record in the block has a key.

If you specify SCALARVARYING, the embedded key must not immediately precede or follow the first byte; hence, the value specified for KEYLOC must be greater than 2.

If you include the KEYLOC option in a VSAM file declaration for checking purposes, and the key location you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

Creating an indexed data set

When you create an indexed data set, you must open the associated file for SEQUENTIAL OUTPUT, and you must present the records in the order of ascending key values. (If there is an error in the key sequence, the KEY condition is raised.) You cannot use a DIRECT file for the creation of an indexed data set.

Table 16 on page 148 shows the statements and options for creating an indexed data set.

You can extend an indexed data set consisting of fixed-length records by adding records sequentially at the end, until the original space allocated for the prime data is filled. You must open the corresponding file for SEQUENTIAL OUTPUT and you must include DISP=MOD in the DD statement.

You can use a single DD statement to define the whole data set (index area, prime area, and overflow area), or you can use two or three statements to define the areas independently. If you use two DD statements, you can define either the index area and the prime area together, or the prime area and the overflow area together.

If you want the entire data set to be on a single volume, there is no advantage to be gained by using more than one DD statement except to define an independent overflow area (see "Overflow area" on page 160). But, if you use separate DD statements to define the index and/or overflow area on volumes separate from that which contains the prime area, you will increase the speed of direct-access to the records in the data set by reducing the number of access mechanism movements required.

When you use two or three DD statements to define an indexed data set, the statements must appear in the order: index area; prime area; overflow area. The first DD statement must have a name (ddname), but the name fields of a second or third DD statement must be blank. The DD statements for the prime and overflow areas must specify the same type of unit (UNIT parameter). You must include all the DCB information for the data set in the first DD statement. DCB=DSORG=IS will suffice in the other statements.

Essential information

To create an indexed data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

You must supply the following information when creating an indexed data set:

- Direct-access device that will write your data set (UNIT or VOLUME parameter of DD statement). Do not request DEFER.
- Block size: You can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size.
- Space requirements: Include space for future needs when you specify the size of the prime, index, and overflow areas. Once you have created an indexed data set, you cannot change its specification.

If you want to keep a direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must specify the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not specify a serial number for a data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing. All the essential parameters required in a DD statement for the creation of an indexed data set are summarized in Table 18 on page 156. Table 19 on page 156 lists the DCB subparameters needed. See the *MVS/370 JCL User's Guide* for a description of the DCB subparameters.

You must request space for the prime data area in the SPACE parameter. You cannot specify a secondary quantity for an indexed data set. Your request must be in units of cylinders unless you place the data set in a specific position on the volume (by specifying a track number in the SPACE parameter). In the latter case, the number of tracks you specify must be equivalent to an integral number of cylinders, and the first track must be the first track of a cylinder other than the first cylinder in the volume.

You can also use the SPACE parameter to specify the amount of space to be used for the cylinder and master indexes (unless you use a separate DD statement for this purpose). If you do not specify the space for the indexes, the operating system will use part of the independent overflow area. If there is no independent overflow area, it will use part of the prime data area.

Table 18. Creating an indexed data set: essential parameters of DD statement

When required	What you must state	Parameters
Always	Output device	UNIT= or VOLUME=REF=
	Storage space required	SPACE=
	Data control block information: see Table 19	DCB=
More than one DD statement	Name of data set and area (index, prime, overflow)	DSNAME=
Data set to be used in another job step but not required at end of job	Disposition	DISP=
Data set to be kept after end of job	Disposition	DISP=
	Name of data set	DSNAME=
Data set to be on particular volume	Volume serial number	VOLUME=SER= or VOLUME=REF=

Table 19 (Page 1 of 2). DCB subparameters for an indexed data set

When required	To specify	Subparameters
These are always required	Record format	RECFM=F, FB, V, or VB
	Block size	BLKSIZE=
	Data set organization	DSORG=IS
	Key length	KEYLEN=
Include at least one of these if overflow is required	Cylinder overflow area and number of tracks per cylinder for overflow records	OPTCD=Y and CYLOFL=
	Independent overflow area	OPTCD=I

Table 19 (Page 2 of 2). DCB subparameters for an indexed data set

When required	To specify	Subparameters
These are optional	Record length	LRECL=
	Embedded key (relative key position)	OPTCD=M
		OPTCD=L
	Master index	
	Automatic processing of dummy records	NTM=
	Number of data management buffers	
	Number of tracks in cylinder index for each master index entry	

Notes:

Full DCB information must appear in the first, or only, DD statement. Subsequent statements require only DSORG=IS.

You must always specify the data set organization (DSORG=IS subparameter of the DCB parameter), and in the first (or only) DD statement you must also specify the length of the key (KEYLEN subparameter of the DCB parameter) unless it is specified in the ENVIRONMENT attribute.

If you want the operating system to recognize dummy records, you must code OPTCD=L in the DCB subparameter of the DD statement. This will cause the operating system to write the dummy identifier in deleted records and to ignore dummy records during sequential read processing. Do not specify OPTCD=L when using blocked or variable-length records with nonembedded keys. If you do this, the dummy record identifier (8)'1'B will overwrite the key of deleted records.

You cannot place an indexed data set on a system output (SYSOUT) device.

Name of the data set

If you use only one DD statement to define your data set, you need not name the data set unless you intend to access it in another job. But if you include two or three DD statements, you must specify a data set name, even for a temporary data set.

The DSNAME parameter in a DD statement that defines an indexed data set not only gives the data set a name, but it also identifies the area of the data set to which the DD statement refers:

```
DSNAME=name(INDEX)
DSNAME=name(PRIME)
DSNAME=name(OVFLOW)
```

If you use one DD statement to define the prime and index or one DD statement to define the prime and overflow area, code DSNAME=name(PRIME). If you use one DD statement for the entire file (prime, index, and overflow), code DSNAME=name(PRIME) or simply DSNAME=name.

Record format and keys

An indexed data set can contain either fixed- or variable-length records, blocked or unblocked. You must always specify the record format, either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (RECFM subparameter).

The key associated with each record can be contiguous with or embedded within the data in the record.

If the records are unblocked, the key of each record is recorded in the data set in front of the record even if it is also embedded within the record, as shown in (a) and (b) of Figure 34 on page 159.

If blocked records do not have embedded keys, the key of each record is recorded within the block in front of the record, and the key of the last record in the block is also recorded just ahead of the block, as shown in (c) of Figure 34.

When blocked records have embedded keys, the individual keys are not recorded separately in front of each record in the block: the key of the last record in the block is recorded in front of the block, as shown in (d) of Figure 34.

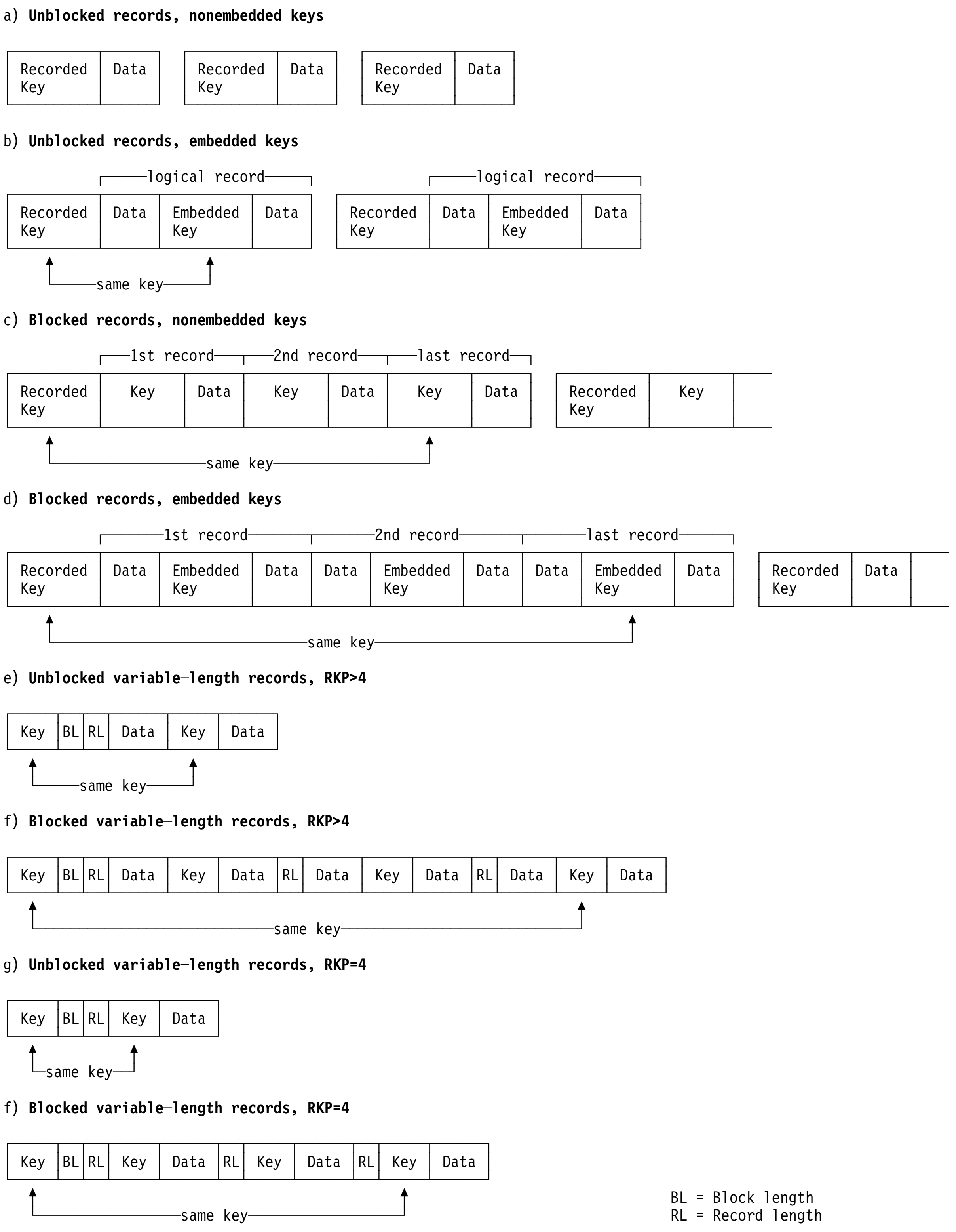

a) Unblocked records, nonembedded keys
Recorded Key
Data
b) Unblocked records, embedded keys
logical record
Recorded Key
Data
Embedded Key
Data
same key
c) Blocked records, nonembedded keys
1st record
2nd record
last record
Recorded Key
Key
Data
same key
d) Blocked records, embedded keys
1st record
2nd record
last record
Recorded Key
Data
Embedded Key
same key
e) Unblocked variable–length records, RKP>4
Key
BL
RL
Data
same key
f) Blocked variable–length records, RKP>4
Key
BL
RL
Data
same key
g) Unblocked variable–length records, RKP=4
Key
BL
RL
Data
same key
f) Blocked variable–length records, RKP=4
Key
BL
RL
Data
same key
BL = Block length
RL = Record length

Figure 34. Record formats in an indexed data set

If you use blocked records with nonembedded keys, the record size that you specify must include the length of the key, and the block size must be a multiple of this combined length. Otherwise, record length and block size refer only to the data in the record. Record format information is shown in Figure 35 on page 160.

If you use records with embedded keys, you must include the DCB subparameter RKP to indicate the position of the key within the record. For fixed-length records the value specified in the RKP subparameter is 1 less than the byte number of the first character of the key. That is, if RKP=1, the key starts in the second byte of the record. The default value if you omit this subparameter is RKP=0, which specifies that the key is not embedded in the record but is separate from it.

For variable-length records, the value you specify in the RKP subparameter must be the relative position of the key within the record plus 4. The extra 4 bytes take into account the 4-byte control field used with variable-length records. For this reason, you must never specify RKP less than 4. When deleting records, you must always specify RKP equal to or greater than 5, since the first byte of the data is used to indicate deletion.

For unblocked records, the key, even if embedded, is always recorded in a position preceding the actual data. Consequently, you do not need to specify the RKP subparameter for unblocked records.

RECORDS	**RKP**	**LRECL**	**BLKSIZE**
Blocked	Not zero	R	R * B
	Zero or omitted	R + K	B*(R+K)
Unblocked	Not zero	R	R
	Zero or omitted	R	R

```
R = Size of data in record
K = Length of keys (as specified in KEYLEN subparameter)
B = Blocking factor

Example:  For blocked records, nonembedded keys, 100 bytes of
          data per record, 10 records per block, key length = 20:

             LRECL=120,BLKSIZE=1200,RECFM=FB
```

Figure 35. Record format information for an indexed data set

Overflow area

If you intend to add records to the data set on a future occasion, you must request either a cylinder overflow area or an independent overflow area, or both.

For a cylinder overflow area, include the DCB subparameter OPTCD=Y and use the subparameter CYLOFL to specify the number of tracks in each cylinder to be reserved for overflow records. A cylinder overflow area has the advantage of a short search time for overflow records, but the amount of space available for overflow records is limited, and much of the space might be unused if the overflow records are not evenly distributed throughout the data set.

For an independent overflow area, use the DCB subparameter OPTCD=I to indicate that overflow records are to be placed in an area reserved for overflow records from all cylinders, and include a separate DD statement to define the overflow area. The use of an independent area has the advantage of reducing the amount of unused space for overflow records, but entails an increased search time for overflow records.

It is good practice to request cylinder overflow areas large enough to contain a reasonable number of additional records and an independent overflow area to be used as the cylinder overflow areas are filled.

If the prime data area is not filled during creation, you cannot use the unused portion for overflow records, nor for any records subsequently added during direct-access (although you can fill the unfilled portion of the last track used). You can reserve space for later use within the prime data area by writing dummy records during creation (see “Dummy records” on page 151).

Master index

If you want the operating system to create a master index for you, include the DCB subparameter OPTCD=M, and indicate in the NTM subparameter the number of tracks in the cylinder index you wish to be referred to by each entry in the master index. The operating system will create up to three levels of master index, the first two levels addressing tracks in the next lower level of the master index.

The creation of a simple indexed data set is illustrated in Figure 36 on page 162. The data set contains a telephone directory, using the subscribers' names as keys to the telephone numbers.

```
//EX8#19  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 TELNOS: PROC OPTIONS(MAIN);
        DCL DIREC FILE RECORD SEQUENTIAL KEYED,
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD,
            NUMBER CHAR(3) DEF CARD POS(21),
            IOFIELD CHAR(3),
            EOF BIT(1) INIT('0'B);
        ON ENDFILE(SYSIN) EOF='1'B;
        OPEN FILE(DIREC) OUTPUT;
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
        IOFIELD=NUMBER;
        WRITE FILE(DIREC) FROM(IOFIELD) KEYFROM(NAME);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;
        CLOSE FILE(DIREC);
 END TELNOS;
/*
//GO.DIREC DD DSN=HPU8.TELNO(INDEX),UNIT=SYSDA,SPACE=(CYL,1),
//            DCB=(RECFM=F,BLKSIZE=3,DSORG=IS,KEYLEN=20,OPTCD=LIY,
//            CYLOFL=2),DISP=(NEW,KEEP)
//         DD DSN=HPU8.TELNO(PRIME),UNIT=SYSDA,SPACE=(CYL,1),
//            DISP=(NEW,KEEP),DCB=DSORG=IS
//         DD DSN=HPU8.TELNO(OVFLOW),UNIT=SYSDA,SPACE=(CYL,1),
//            DISP=(NEW,KEEP),DCB=DSORG=IS
//GO.SYSIN DD *
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,D.        141
CORY,G.             336
ELLIOTT,D.          875
FIGGINS,S.          413
HARVEY,C.D.W.       205
HASTINGS,G.M.       391
KENDALL,J.G.        294
LANCASTER,W.R.      624
MILES,R.            233
NEWMAN,M.W.         450
PITT,W.H.           515
ROLF,D.E.           114
SHEERS,C.D.         241
SUTCLIFFE,M.        472
TAYLOR,G.C.         407
WILTON,L.W.         404
WINSTONE,E.M.       307
/*
```

Figure 36. Creating an indexed data set

Accessing and updating an indexed data set

Once you create an indexed data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. In the case of F-format records, you can also open it for OUTPUT to add records at the end of the data set. The keys for these records must have higher values than the existing keys for that data set and must be in ascending order. Table 16 on page 148 shows the statements and options for accessing an indexed data set.

Sequential input allows you to read the records in ascending key sequence, and in sequential update you can read and rewrite each record in turn. Using direct input, you can read records using the READ statement, and in direct update you can read or delete existing records or add new ones. Sequential and direct-access are discussed in further detail below.

Using sequential access

You can open a sequential file that is used to access an indexed data set with either the INPUT or the UPDATE attribute. You do not need to include source keys in the data transmission statements, nor do you need to give the file the KEYED attribute. Sequential access is in order of ascending recorded-key values. Records are retrieved in this order, and not necessarily in the order in which they were added to the data set. Dummy records are not retrieved if you include the subparameter OPTCD=L in the DD statement that defines the data set.

Rules governing the relationship between the READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses an indexed data set are identical to those for a consecutive data set (described in Chapter 7, "Defining and using consecutive data sets" on page 123).

You must not alter embedded keys in a record to be updated. The modified record must always overwrite the update record in the data set.

Additionally, records can be effectively deleted from the data set. Using a DELETE statement marks a record as a dummy by putting (8)'1'B in the first byte. You should not use the DELETE statement to process a data set with F-format blocked records and either KEYLOC=1 or RKP=0, or a data set with V- or VB-format records and either KEYLOC=1 or RKP=4. (The code (8)'1'B would overwrite the first byte of the recorded key.)

You can position INDEXED KEYED files opened for SEQUENTIAL INPUT and SEQUENTIAL UPDATE to a particular record within the data set by using either a READ KEY or a DELETE KEY operation that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option access the next records in the data set sequentially. A subsequent READ statement without the KEY option causes the record with the next higher recorded key to be read (even if the keyed record has not been found).

Define the length of the recorded keys in an indexed data set with the KEYLENGTH ENVIRONMENT option or the KEYLEN subparameter of the DD statement that defines the data set. If the length of a source key is greater than the specified length of the recorded keys, the source key is truncated on the right.

The effect of supplying a source key that is shorter than the recorded keys in the data set differs according to whether or not you specify the GENKEY option in the ENVIRONMENT attribute. In the absence of the GENKEY option, the source key is padded on the right with blanks to the length you specify in the KEYLENGTH option of the ENVIRONMENT attribute, and the record with this padded key is read (if such a record exists). If you specify the GENKEY option, the source key is interpreted as a generic key, and the first record with a key in the class identified by this generic key is read. (For further details, see "GENKEY option — key classification" on page 112.)

Using direct access

You can open a direct file that is used to access an indexed data set with either the INPUT or the UPDATE attribute. You must include source keys in all data transmission statements; the DIRECT attribute implies the KEYED attribute.

You can use a DIRECT UPDATE file to retrieve, add, delete, or replace records in an indexed data set according to the following conventions:

Retrieval
If you include the subparameter OPTCD=L in the DD statement that defines the data set, dummy records are not made available by a READ statement (the KEY condition is raised).

Addition
A WRITE statement that includes a unique key causes a record to be inserted into the data set. If the key is the same as the recorded key of a dummy record, the new record replaces the dummy record. If the key is the same as the recorded key of a record that is not marked as deleted, or if there is no space in the data set for the record, the KEY condition is raised.

Deletion
The record specified by the source key in a DELETE statement is retrieved, marked as deleted, and rewritten into the data set. The effect of the DELETE statement is to insert the value (8)'1'B in the first byte of the data in a record. Deletion is possible only if you specify OPTCD=L in the DD statement that defines the data set when you create it. If the data set has F-format blocked records with RKP=0 or KEYLOC=1, or V-format records with RKP=4 or KEYLOC=1, records cannot be deleted. (The code (8)'1'B would overwrite the embedded keys.)

Replacement
The record specified by a source key in a REWRITE statement is replaced by the new record. If the data set contains F-format blocked records, a record replaced with a REWRITE statement causes an implicit READ statement to be executed unless the previous I/O statement was a READ statement that obtained the record to be replaced. If the data set contains V-format records and the updated record has a length different from that of the record read, the whole of the remainder of the track will be removed, and can cause data to be moved to an overflow track.

Essential information

To access an indexed data set, you must define it in one, two, or three DD statements. The DD statements must correspond with those used when the data set is created. The following paragraphs indicate the essential information you must include in each DD statement. Table 20 on page 165 summarizes this information.

Table 20. Accessing an indexed data set: essential parameters of DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	
DCB=	Data control block information	
UNIT= or VOLUME=REF=	Input device	If data set not cataloged
VOLUME=SER=	Volume serial number	

If the data set is cataloged, you need supply only the following information in each DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information that describes the data set in the system catalog and, if necessary, will request the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will process the data set and give the serial number of the volume that contains it (UNIT and VOLUME parameters).

Example

The program in Figure 37 on page 166 updates the data set of the previous example (Figure 36 on page 162) and prints out its new contents. The input data includes the following codes to indicate the operations required:

A	Add a new record.
C	Change an existing record.
D	Delete an existing record.

```
//EX8#20  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN     DD *
 DIRUPDT: PROC OPTIONS(MAIN);
         DCL DIREC FILE RECORD KEYED,
             NUMBER CHAR(3),NAME CHAR(20),CODE CHAR(1),ONCODE BUILTIN,
             EOF BIT(1) INIT('0'B);
         ON ENDFILE(SYSIN) EOF='1'B;
         ON KEY(DIREC) BEGIN;
          IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                           ('NOT FOUND:',NAME)(A(15),A);
          IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                           ('DUPLICATE:',NAME)(A(15),A);
          END;
         OPEN FILE(DIREC) DIRECT UPDATE;
         GET FILE(SYSIN) EDIT(NAME,NUMBER,CODE)
           (COLUMN(1),A(20),A(3),A(1));
         DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
            (A(1),A(20),A(1),A(3),A(1),A(1));
         SELECT (CODE);
            WHEN('A') WRITE FILE(DIREC) FROM(NUMBER) KEYFROM(NAME);
            WHEN('C') REWRITE FILE(DIREC) FROM(NUMBER) KEY(NAME);
            WHEN('D') DELETE FILE(DIREC) KEY(NAME);
            OTHERWISE PUT FILE(SYSPRINT) SKIP
              EDIT('INVALID CODE:',NAME)(A(15),A);
         END;
         GET FILE(SYSIN) EDIT(NAME,NUMBER,CODE)
         (COLUMN(1),A(20),A(3),A(1));
         END;
         CLOSE FILE(DIREC);
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(DIREC) SEQUENTIAL INPUT;
         EOF='0'B;
         ON ENDFILE(DIREC) EOF='1'B;
         READ FILE(DIREC) INTO(NUMBER) KEYTO(NAME);
         DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT(NAME,NUMBER)(A);
         READ FILE(DIREC) INTO(NUMBER) KEYTO(NAME);
         END;
         CLOSE FILE(DIREC);       END DIRUPDT;
/*
//GO.DIREC DD DSN=HPU8.TELNO(INDEX),DISP=(OLD,DELETE),
//     VOL=SER=nnnnnn,UNIT=SYSDA
//         DD DSN=HPU8.TELNO(PRIME),DISP=(OLD,DELETE),
//     VOL=SER=nnnnnn,UNIT=SYSDA
//         DD DSN=HPU8.TELNO(OVFLOW),DISP=(OLD,DELETE),
//     VOL=SER=nnnnnn,UNIT=SYSDA
//GO.SYSIN DD *
NEWMAN,M.W.         516C
GOODFELLOW,D.T.     889A
MILES,R.               D
HARVEY,C.D.W.       209A
BARTLETT,S.G.       183A
CORY,G.                D
READ,K.M.           001A
PITT,W.H.
ROLF,D.E.              D
ELLIOTT,D.          291C
HASTINS,G.M.           D
BRAMLEY,O.H.        439
/*
```

Figure 37. Updating an indexed data set

Reorganizing an indexed data set

It is necessary to reorganize an indexed data set periodically because the addition of records to the data set results in an increasing number of records in the overflow area. Therefore, even if the overflow area does not eventually become full, the average time required for the direct retrieval of a record will increase. The frequency of reorganization depends on how often you update the data set, on how much storage is available in the data set, and on your timing requirements.

Reorganizing the data set also eliminates records that are marked as “deleted” but are still present within the data set.

There are two ways to reorganize an indexed data set:

- Read the data set into an area of main storage or onto a temporary consecutive data set, and then recreate it in the original area of auxiliary storage.
- Read the data set sequentially and write it into a new area of auxiliary storage. You can then release the original auxiliary storage.

Chapter 9. Defining and using regional data sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. How to create and access regional data sets for each type of regional organization is then discussed.

A data set with regional organization is divided into regions, each of which is identified by a region number, and each of which can contain one record or more than one record, depending on the type of regional organization. The regions are numbered in succession, beginning with zero, and a record can be accessed by specifying its region number, and perhaps a key, in a data transmission statement.

Regional data sets are confined to direct-access devices.

Regional organization of a data set allows you to control the physical placement of records in the data set, and to optimize the access time for a particular application. Such optimization is not available with consecutive or indexed organization, in which successive records are written either in strict physical sequence or in logical sequence depending on ascending key values; neither of these methods takes full advantage of the characteristics of direct-access storage devices.

You can create a regional data set in a manner similar to a consecutive or indexed data set, presenting records in the order of ascending region numbers; alternatively, you can use direct-access, in which you present records in random sequence and insert them directly into preformatted regions. Once you create a regional data set, you can access it by using a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. You do not need to specify either a region number or a key if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Records within a regional data set are either actual records containing valid data or dummy records. The nature of the dummy records depends on the type of regional organization; the three types of regional organization are described below.

The major advantage of regional organization over other types of data set organization is that it allows you to control the relative placement of records; by judicious programming, you can optimize record access in terms of device capabilities and the requirements of particular applications.

Direct access of regional data sets is quicker than that of indexed data sets, but regional data sets have the disadvantage that sequential processing can present records in random sequence; the order of sequential retrieval is not necessarily that in which the records were presented, nor need it be related to the relative key values.

Table 21 on page 169 lists the data transmission statements and options that you can use to create and access a regional data set.

Table 21 (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	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	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	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference) FROM(reference);	
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	

Table 21 (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 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);	

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);
	
	UNLOCK FILE(file-reference) KEY(expression);	

Notes:

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);
3. The file must not have the UPDATE attribute when creating new data sets.

Defining files for a regional data set

Use a file declaration with the following attributes to define a sequential regional data set:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

To define a direct regional data set, use a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 10 on page 108. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to regional data sets are:

```
REGIONAL({1})
F|V|VS|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
KEYLENGTH(n)
```

REGIONAL option

Use the REGIONAL option to define a file with regional organization.

```
►►──REGIONAL──(──1──)──────────────────────────────────────────►◄
```

1 specifies REGIONAL(1)

REGIONAL(1)
specifies that the data set contains F-format records that do not have recorded keys. Each region in the data set contains only one record; therefore, each region number corresponds to a relative record within the data set (that is, region numbers start with 0 at the beginning of the data set).

Although REGIONAL(1) data sets have no recorded keys, you can use REGIONAL(1) DIRECT INPUT or UPDATE files to process data sets that do have recorded keys.

REGIONAL(1) organization is most suited to applications where there are no duplicate region numbers, and where most of the regions will be filled (reducing wasted space in the data set).

Using keys with REGIONAL data sets

There are two kinds of keys, recorded keys and source keys. A *recorded key* is a character string that immediately precedes each record in the data set to identify that record; its length cannot exceed 255 characters. A *source key* is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. When you access a record in a regional data set, the source key gives a region number, and can also give a recorded key.

You specify the length of the recorded keys in a regional data set with the KEYLENGTH option of the ENVIRONMENT attribute, or the KEYLEN subparameter on the DD statement. Unlike the keys for indexed data sets, recorded keys in a regional data set are never embedded within the record.

Using REGIONAL(1) data sets

In a REGIONAL(1) data set, since there are no recorded keys, the region number serves as the sole identification of a particular record. The character value of the source key should represent an unsigned decimal integer that should not exceed 16777215 (although the actual number of records allowed can be smaller, depending on a combination of record size, device capacity, and limits of your access method. For direct regional(1) files with fixed format records, the maximum number of tracks which can be addressed by relative track addressing is 65,536.) If the region number exceeds this figure, it is treated as modulo 16777216; for instance, 16777226 is treated as 10. Only the characters 0 through 9 and the blank character are valid in the source key; leading blanks are interpreted as zeros. Embedded blanks are not allowed in the number; the first embedded blank, if any, terminates the region number. If more than 8 characters appear in the source key, only the rightmost 8 are used as the region number; if there are fewer than 8 characters, blanks (interpreted as zeros) are inserted on the left.

Dummy Records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant (8)'1'B in its first byte. Although such dummy records are inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read; your PL/I program must be prepared to recognize them. You can replace dummy records with valid data. Note that if you insert (8)'1'B in the first byte, the record can be lost if you copy the file onto a data set that has dummy records that are not retrieved.

Creating a REGIONAL(1) data set

You can create a REGIONAL(1) data set either sequentially or by direct-access. Table 21 on page 169 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, the opening of the file causes all tracks on the data set to be cleared, and a capacity record to be written at the beginning of each track to record the amount of space available on that track. You must present records in ascending order of region numbers; any region you omit from the sequence is filled with a dummy record. If there is an error in the sequence, or if you present a duplicate key, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If you 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.

If you use a DIRECT OUTPUT file to create the data set, the whole primary extent allocated to the data set is filled with dummy records when the file is opened. You can present records in random order; if you present a duplicate, the existing record will be overwritten.

For sequential creation, the data set can have up to 15 extents, which can be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

Example

Creating a REGIONAL(1) data set is illustrated in Figure 38. The data set is a list of telephone numbers with the names of the subscribers to whom they are allocated. The telephone numbers correspond with the region numbers in the data set, the data in each occupied region being a subscriber's name.

```
//EX9    JOB
//STEP1  EXEC IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 CRR1:   PROC OPTIONS(MAIN);
  /*  CREATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY     */

   DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
   DCL  SYSIN FILE INPUT RECORD;
   DCL  SYSIN_REC BIT(1) INIT('1'B);
   DCL 1   CARD,
        2   NAME   CHAR(20),
        2   NUMBER CHAR( 2),
        2   CARD_1 CHAR(58);
   DCL IOFIELD CHAR(20);

      ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
      OPEN FILE(NOS);
      READ FILE(SYSIN) INTO(CARD);

      DO WHILE(SYSIN_REC);
         IOFIELD = NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         READ FILE(SYSIN) INTO(CARD);
      END;

      CLOSE FILE(NOS);
  END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS     DD DSN=NOS,UNIT=SYSDA,SPACE=(20,100),
//           DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.       28
CHEESNAME,L.       11
CORY,G.            36
ELLIOTT,D.         85
FIGGINS,E.S.       43
HARVEY,C.D.W.      25
HASTINGS,G.M.      31
KENDALL,J.G.       24
LANCASTER,W.R.     64
MILES,R.           23
NEWMAN,M.W.        40
PITT,W.H.          55
ROLF,D.E.          14
SHEERS,C.D.        21
SURCLIFFE,M.       42
TAYLOR,G.C.        47
WILTON,L.W.        44
WINSTONE,E.M.      37
/*
```

Figure 38. Creating a REGIONAL(1) data set

Accessing and updating a REGIONAL(1) data set

Once you create a REGIONAL(1) data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. You can open it for OUTPUT only if the existing data set is to be overwritten. Table 21 on page 169 shows the statements and options for accessing a regional data set.

Sequential access

To open a SEQUENTIAL file that is used to process a REGIONAL(1) data set, use either the INPUT or UPDATE attribute. You must not include the KEY option in data transmission statements, but the file can have the KEYED attribute, since you can use the KEYTO option. If the target character string referenced in the KEYTO option has more than 8 characters, the value returned (the 8-character region number) is padded on the left with blanks. If the target string has fewer than 8 characters, the value returned is truncated on the left.

Sequential access is in the order of ascending region numbers. All records are retrieved, whether dummy or actual, and you must ensure that your PL/I program recognizes dummy records.

Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region-number sequence, and in sequential update you can read and rewrite each record in turn.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(1) data set are identical to those for a consecutive data set. Consecutive data sets are discussed in detail in Chapter 7, “Defining and using consecutive data sets” on page 123.

Direct access

To open a DIRECT file that is used to process a REGIONAL(1) data set you can use either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

Use DIRECT UPDATE files to retrieve, add, delete, or replace records in a REGIONAL(1) data set according to the following conventions:

Retrieval	All records, whether dummy or actual, are retrieved. Your program must recognize dummy records.
Addition	A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key.
Deletion	The record you specify by the source key in a DELETE statement is converted to a dummy record.
Replacement	The record you specify by the source key in a REWRITE statement, whether dummy or actual, is replaced.

Example

Updating a REGIONAL(1) data set is illustrated in Figure 39 on page 176. Like the program in Figure 37 on page 166, this program updates the data set and lists its contents. Before each new or updated record is written, the existing record in the region is tested to ensure that it is a dummy; this is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

```
//EX10    JOB
//STEP2   EXEC  IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN  DD  *
 ACR1: PROC OPTIONS(MAIN);
   /*  UPDATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY      */
    DCL NOS FILE RECORD  KEYED ENV(REGIONAL(1));
    DCL  SYSIN FILE INPUT RECORD;
    DCL  (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
    DCL 1   CARD,
         2   NAME   CHAR(20),
         2   (NEWNO,OLDNO)  CHAR( 2),
         2   CARD_1 CHAR( 1),
         2   CODE   CHAR( 1),
         2   CARD_2 CHAR(54);
    DCL IOFIELD CHAR(20);
    DCL BYTE    CHAR(1) DEF IOFIELD;

   ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
   OPEN FILE (NOS) DIRECT UPDATE;
   READ FILE(SYSIN) INTO(CARD);

   DO WHILE(SYSIN_REC);
      SELECT(CODE);
         WHEN('A','C') DO;
             IF CODE = 'C' THEN
                DELETE FILE(NOS) KEY(OLDNO);
             READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
             IF UNSPEC(BYTE) = (8)'1'B
                THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
             ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
         END;
         WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
      OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
      END;
      READ FILE(SYSIN) INTO(CARD);
   END;

   CLOSE FILE(SYSIN),FILE(NOS);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(NOS) SEQUENTIAL INPUT;
   ON ENDFILE(NOS) NOS_REC = '0'B;
   READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   DO WHILE(NOS_REC);
      IF UNSPEC(BYTE) ¬= (8)'1'B
         THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD)(A(2),X(3),A);
      PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   END;
  CLOSE FILE(NOS);
  END ACR1;
/*
//GO.NOS   DD DSN=J44PLI.NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.         5640 C
GOODFELLOW,D.T.     89   A
MILES,R.              23 D
HARVEY,C.D.W.       29   A
BARTLETT,S.G.       13   A
CORY,G.               36 D
READ,K.M.           01   A
PITT,W.H.             55
ROLF,D.F.             14 D
ELLIOTT,D.          4285 C
HASTINGS,G.M.         31 D
BRAMLEY,O.H.        4928 C
/*
```

Figure 39. Updating a REGIONAL(1) data set

Essential information for creating and accessing regional data sets

To create a regional data set, you must give the operating system certain information, either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

You must supply the following information when creating a regional data set:

- Device that will write your data set (UNIT or VOLUME parameter of DD statement).
- Block size: You can specify the block size either in your PL/I program (in the BLKSIZE option of the ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size.

If you want to keep a data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but do not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system allocates one, informs the operator, and prints the number on your program listing. All the essential parameters required in a DD statement for the creation of a regional data set are summarized in Table 22; and Table 23 on page 178 lists the DCB subparameters needed. See your *OS/390 JCL User's Guide* for a description of the DCB subparameters.

You cannot place a regional data set on a system output (SYSOUT) device.

In the DCB parameter, you must always specify the data set organization as direct by coding DSORG=DA. You cannot specify the DUMMY or DSN=NULLFILE parameters in a DD statement for a regional data set.

Table 22 (Page 1 of 2). Creating a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
UNIT= or VOLUME=REF=	Output device[1]	Always
SPACE=	Storage space required[2]	
DCB=	Data control block information: see Table 23 on page 178	
DISP=	Disposition	Data set to be used in another job step but not required in another job
DISP=	Disposition	Data set to be kept after end of job
DSNAME=	Name of data set	

Table 22 (Page 2 of 2). Creating a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
VOLUME=SER= or VOLUME=REF=	Volume serial number	Data set to be on particular volume

1Regional data sets are confined to direct-access devices.

2For sequential access, the data set can have up to 15 extents, which can be on more than one volume. For creation with DIRECT access, the data set can have only one extent.

To access a regional data set, you must identify it to the operating system in a DD statement. The following paragraphs indicate the minimum information you must include in the DD statement; this information is summarized in Table 24.

If the data set is cataloged, you only need to supply the following information in your DD statement:

- The name of the data set (DSNAME parameter). The operating system locates the information that describes the data set in the system catalog and, if necessary, requests the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

Unlike indexed data sets, regional data sets do not require the subparameter OPTCD=L in the DD statement.

When opening a multiple-volume regional data set for sequential update, the ENDFILE condition is raised at the end of the first volume.

Table 23. DCB subparameters for a regional data set

Subparameters	To specify	When required
RECFM=F	Record format1	These are always required
BLKSIZE=	Block size1	
DSORG=DA	Data set organization	

1Or you can specify the block size in the ENVIRONMENT attribute.

Table 24. Accessing a regional data set: essential parameters of the DD statement

Parameters	What you must state	When required
DSNAME=	Name of data set	Always
DISP=	Disposition of data set	
UNIT= or VOLUME=REF=	Input device	If data set not cataloged
VOLUME=SER=	Volume serial number	

Chapter 10. Defining and using VSAM data sets

This chapter covers VSAM (the Virtual Storage Access Method) organization for record-oriented data transmission, VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and the statements you use to load and access the three types of VSAM data sets that PL/I supports—entry-sequenced, key-sequenced, and relative record. The chapter is concluded by a series of examples showing the PL/I statements, Access Method Services commands, and JCL statements necessary to create and access VSAM data sets.

For additional information about the facilities of VSAM, the structure of VSAM data sets and indexes, the way in which they are defined by Access Method Services, and the required JCL statements, see the VSAM publications for your system.

Using VSAM data sets

How to run a program with VSAM data sets

Before you execute a program that accesses a VSAM data set, you need to know:

- The name of the VSAM data set
- The name of the PL/I file
- Whether you intend to share the data set with other users

Then you can write the required DD statement to access the data set:

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

For example, if your file is named PL1FILE, your data set named VSAMDS, and you want exclusive control of the data set, enter:

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

To share your data set, use DISP=SHR.

To optimize VSAM's performance by controlling the number of VSAM buffers used for your data set, see the VSAM publications.

VSAM organization

PL/I provides support for three types of VSAM data sets:

- Key-sequenced data sets (KSDS)
- Entry-sequenced data sets (ESDS)
- Relative record data sets (RRDS).

These correspond roughly to PL/I indexed, consecutive, and regional data set organizations, respectively. They are all ordered, and they can all have keys associated with their records. Both sequential and keyed access are possible with all three types.

Although only key-sequenced data sets have keys as part of their logical records, keyed access is also possible for entry-sequenced data sets (using relative-byte addresses) and relative record data sets (using relative record numbers).

All VSAM data sets are held on direct-access storage devices, and a virtual storage operating system is required to use them.

The physical organization of VSAM data sets differs from those used by other access methods. VSAM does not use the concept of blocking, and, except for relative record data sets, records need not be of a fixed length. In data sets with VSAM organization, the data items are arranged in *control intervals*, which are in turn arranged in *control areas*. For processing purposes, the data items within a control interval are arranged in logical records. A control interval can contain one or more logical records, and a logical record can span two or more control intervals. Concern about blocking factors and record length is largely removed by VSAM, although records cannot exceed the maximum specified size. VSAM allows access to the control intervals, but this type of access is not supported by PL/I.

VSAM data sets have prime indexes. A *prime index* is the index to a KSDS that is established when you define a data set; it always exists and can be the only index for a KSDS. The prime index can never have duplicate keys.

Before using a VSAM data set for the first time, you need to define it to the system with the DEFINE command of Access Method Services, which you can use to completely define the type, structure, and required space of the data set. This command also defines the data set's indexes (together with their key lengths and locations) and the index upgrade set if the data set is a KSDS. A VSAM data set is thus “created” by Access Method Services.

The operation of writing the initial data into a newly created VSAM data set is referred to as *loading* in this publication.

Use the three different types of data sets according to the following purposes:

- Use *entry-sequenced data sets* for data that you primarily access in the order in which it was created (or the reverse order).
- Use *key-sequenced data sets* when you normally access records through keys within the records (for example, a stock-control file where the part number is used to access a record).
- Use *relative record data sets* for data in which each item has a particular number, and you normally access the relevant record by that number (for example, a telephone system with a record associated with each number).

You can access records in all types of VSAM data sets either directly by means of a key, or sequentially (backward or forward). You can also use a combination of the two ways: Select a starting point with a key and then read forward or backward from that point.

Table 25 on page 181 shows how the same data could be held in the three different types of VSAM data sets and illustrates their respective advantages and disadvantages.

Table 25. Types and advantages of VSAM data sets

Data set type	Method of loading	Method of reading	Method of updating	Pros and cons
Key-Sequenced	Sequentially in order or prime index which must be unique	KEYED by specifying key of record in prime index SEQUENTIAL backward or forward in order of any index Positioning by key followed by sequential reading either backward or forward	KEYED specifying a unique key in any index SEQUENTIAL following positioning by unique key Record deletion allowed Record insertion allowed	***Advantages***: Complete access and updating ***Disadvantages***: Records must be in order of prime index before loading ***Uses***: For uses where access will be related to key
Entry-Sequenced	Sequentially (forward only) The RBA of each record can be obtained and used as a key	SEQUENTIAL backward or forward KEYED using RBA Positioning by key followed by sequential either backward or forward	New records at end only Existing records cannot have length changed Record deletion not allowed	***Advantages***: Simple fast creation No requirement for a unique index ***Disadvantages***: Limited updating facilities ***Uses***: For uses where data will primarily be accessed sequentially
Relative Record	Sequentially starting from slot 1 KEYED specifying number of slot Positioning by key followed by sequential writes	KEYED specifying numbers as key Sequential forward or backward omitting empty records	Sequentially starting at a specified slot and continuing with next slot Keyed specifying numbers as key Record deletion allowed Record insertion into empty slots allowed	***Advantages***: Speedy access to record by number ***Disadvantages***: Structure tied to numbering sequences Fixed length records ***Uses***: For use where records will be accessed by number

Keys for VSAM data sets

All VSAM data sets can have keys associated with their records. For key-sequenced data sets, the key is a defined field within the logical record. For entry-sequenced data sets, the key is the *relative byte address* (RBA) of the record. For relative-record data sets, the key is a *relative record number*.

Keys for indexed VSAM data sets

Keys for key-sequenced data sets are part of the logical records recorded on the data set. You define the length and location of the keys when you create the data set.

The ways you can reference the keys in the KEY, KEYFROM, and KEYTO options are as described under “KEY(expression) Option,” “KEYFROM(expression) Option,” and “KEYTO(reference) Option” in Chapter 12 of the *PL/I Language Reference*. See also “Using keys” on page 147.

Relative byte addresses (RBA)

Relative byte addresses allow you to use keyed access on an ESDS associated with a KEYED SEQUENTIAL file. The RBAs, or keys, are character strings of length 4, and their values are defined by VSAM. You cannot construct or manipulate RBAs in PL/I; you can, however, compare their values in order to determine the relative positions of records within the data set. RBAs are not normally printable.

You can obtain the RBA for a record by using the KEYTO option, either on a WRITE statement when you are loading or extending the data set, or on a READ statement when the data set is being read. You can subsequently use an RBA obtained in either of these ways in the KEY option of a READ or REWRITE statement.

Do not use an RBA in the KEYFROM option of a WRITE statement.

VSAM allows use of the relative byte address as a key to a KSDS, but this use is not supported by PL/I.

Relative record numbers

Records in an RRDS are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record. You can use these relative record numbers as keys for keyed access to the data set.

Keys used as relative record numbers are character strings of length 8. The character value of a source key you use in the KEY or KEYFROM option must represent an unsigned integer. If the source key is not 8 characters long, it is truncated or padded with blanks (interpreted as zeros) on the left. The value returned by the KEYTO option is a character string of length 8, with leading zeros suppressed.

Choosing a data set type

When planning your program, the first decision to be made is which type of data set to use. There are three types of VSAM data sets and five types of non-VSAM data sets available to you. VSAM data sets can provide all the function of the other types of data sets, plus additional function available only in VSAM. VSAM can usually match other data set types in performance, and often improve upon it. However, VSAM is more subject to performance degradation through misuse of function.

The comparison of all eight types of data sets given in Table 11 on page 115 is helpful; however, many factors in the choice of data set type for a large installation are beyond the scope of this book.

When choosing between the VSAM data set types, you should base your choice on the most common sequence in which you will require your data. The following is a suggested procedure that you can use to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and how it will be accessed.
   a. Primarily sequentially — favors ESDS.
   b. Primarily by key — favors KSDS.
   c. Primarily by number — favors RRDS.

2. Determine how you will load the data set. Note that you must load a KSDS in key sequence.
3. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported. Figure 40 might be helpful.

Do not try to access a dummy VSAM data set, because you will receive an error message indicating that you have an undefined file.

Table 26 on page 186, Table 27 on page 189, and Table 28 on page 195 show the statements allowed for entry-sequenced data sets, indexed data sets, and relative record data sets, respectively.

	SEQUENTIAL	**KEYED SEQUENTIAL**	**DIRECT**
INPUT	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	
OUTPUT	ESDS	ESDS	KSDS
	RRDS	KSDS	RRDS
		RRDS	
UPDATE	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	

Key:	ESDS	Entry-sequenced data set
	KSDS	Key-sequenced data set
	RRDS	Relative record data set

You can combine the attributes on the left with those at the top of the figure for the data sets and paths shown. For example, only an ESDS and an RRDS can be SEQUENTIAL OUTPUT.

PL/I does not support dummy VSAM data sets.

Figure 40. VSAM data sets and allowed file attributes

Defining files for VSAM data sets

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
            [KEYED]
             ENVIRONMENT(options);
```

Table 10 on page 108 shows the default attributes. The file attributes are described in the *PL/I Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Some combinations of the file attributes INPUT or OUTPUT or UPDATE and DIRECT or SEQUENTIAL or KEYED SEQUENTIAL are allowed only for certain types of VSAM data sets. Figure 40 on page 183 shows the compatible combinations.

Specifying ENVIRONMENT options

Many of the options of the ENVIRONMENT attribute affecting data set structure are not needed for VSAM data sets. If you specify them, they are either ignored or are used for checking purposes. If those that are checked conflict with the values defined for the data set, the UNDEFINEDFILE condition is raised when an attempt is made to open the file.

The ENVIRONMENT options applicable to VSAM data sets are:

```
BKWD
GENKEY
REUSE
SCALARVARYING
VSAM
```

GENKEY and SCALARVARYING options have the same effect as they do when you use them for non-VSAM data sets.

The options that are checked for a VSAM data set are RECSIZE and, for a key-sequenced data set, KEYLENGTH and KEYLOC. Table 10 on page 108 shows which options are ignored for VSAM. Table 10 on page 108 also shows the required and default options.

For VSAM data sets, you specify the maximum and average lengths of the records to the Access Method Services utility when you define the data set. If you include the RECSIZE option in the file declaration for checking purposes, specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

BKWD option

Use the BKWD option to specify backward processing for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file associated with a VSAM data set.

```
►►──BKWD──────────────────────────────────────────────────────►◄
```

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

```
A B C1 C2 C3 D E
```

where C1, C2, and C3 have the same key, they are recovered in the sequence:

```
E D C1 C2 C3 B A
```

When a file with the BKWD option is opened, the data set is positioned at the last record. ENDFILE is raised in the normal way when the start of the data set is reached.

Do not specify the BKWD option with either the REUSE option or the GENKEY option. Also, the WRITE statement is not allowed for files declared with the BKWD option.

GENKEY option

For the description of this option, see “GENKEY option — key classification” on page 112.

REUSE option

Use the REUSE option to specify that an OUTPUT file associated with a VSAM data set is to be used as a work file.

```
►►──REUSE──────────────────────────────────────────────────────────►◄
```

The data set is treated as an empty data set each time the file is opened. Any secondary allocations for the data set are released, and the data set is treated exactly as if it were being opened for the first time.

Do not associate a file that has the REUSE option with a data set that has the BKWD option, and do not open it for INPUT or UPDATE.

The REUSE option takes effect only if you specify REUSE in the Access Method Services DEFINE CLUSTER command.

VSAM option

Specify the VSAM option for VSAM data sets.

```
►►──VSAM───────────────────────────────────────────────────────────►◄
```

Performance options

You can specify the buffer options in the AMP parameter of the DD statement; they are explained in your Access Method Services manual.

Defining VSAM data sets

Use the DEFINE CLUSTER command of Access Method Services to define and catalog VSAM data sets. To use the DEFINE command, you need to know:

- The name and password of the master catalog if the master catalog is password protected
- The name and password of the VSAM private catalog you are using if you are not using the master catalog
- Whether VSAM space for your data set is available

- The type of VSAM data set you are going to create
- The volume on which your data set is to be placed
- The average and maximum record size in your data set
- The position and length of the key for an indexed data set
- The space to be allocated for your data set
- How to code the DEFINE command
- How to use the Access Method Services program.

When you have the information, you are in a position to code the DEFINE command and then define and catalog the data set using Access Method Services.

Entry-sequenced data sets

The statements and options allowed for files associated with an ESDS are shown in Table 26.

Table 26. Statements and options allowed for loading and accessing VSAM entry-sequenced data sets

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

Notes:

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use either of the options KEY or KEYTO, it must also include the attribute KEYED.
2. The statement "READ FILE(file-reference);" is equivalent to the statement "READ FILE(file-reference) IGNORE (1);."
3. The expression used in the KEY option must be a relative byte address, previously obtained by means of the KEYTO option.

8/99 - Prepare for next release

Loading an ESDS

When an ESDS is being loaded, the associated file must be opened for SEQUENTIAL OUTPUT. The records are retained in the order in which they are presented.

You can use the KEYTO option to obtain the relative byte address of each record as it is written. You can subsequently use these keys to achieve keyed access to the data set.

Using a SEQUENTIAL file to access an ESDS

You can open a SEQUENTIAL file that is used to access an ESDS with either the INPUT or the UPDATE attribute. If you use either of the options KEY or KEYTO, the file must also have the KEYED attribute.

Sequential access is in the order that the records were originally loaded into the data set. You can use the KEYTO option on the READ statements to recover the RBAs of the records that are read. If you use the KEY option, the record that is recovered is the one with the RBA you specify. Subsequent sequential access continues from the new position in the data set.

For an UPDATE file, the WRITE statement adds a new record at the end of the data set. With a REWRITE statement, the record rewritten is the one with the specified RBA if you use the KEY option; otherwise, it is the record accessed on the previous READ. You must not attempt to change the length of the record that is being replaced with a REWRITE statement.

The DELETE statement is not allowed for entry-sequenced data sets.

Defining and loading an ESDS

In Figure 41 on page 188, the data set is defined with the DEFINE CLUSTER command and given the name PLIVSAM.AJC1.BASE. The NONINDEXED keyword causes an ESDS to be defined.

The PL/I program writes the data set using a SEQUENTIAL OUTPUT file and a WRITE FROM statement. The DD statement for the file contains the DSNAME of the data set given in the NAME parameter of the DEFINE CLUSTER command.

The RBA of the records could have been obtained during the writing for subsequent use as keys in a KEYED file. To do this, a suitable variable would have to be declared to hold the key and the WRITE...KEYTO statement used. For example:

```
DCL CHARS CHAR(4);
WRITE FILE(FAMFILE) FROM (STRING)
 KEYTO(CHARS);
```

Note that the keys would not normally be printable, but could be retained for subsequent use.

The cataloged procedure IBMZCBG is used. Because the same program (in Figure 41 on page 188) can be used for adding records to the data set, it is retained in a library. This procedure is shown in the next example.

```
//OPT9#7  JOB
//STEP1    EXEC   PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD    *
     DEFINE CLUSTER -
      (NAME(PLIVSAM.AJC1.BASE) -
      VOLUMES(nnnnnn) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2))
/*
//STEP2  EXEC IBMZCLG
//PLI.SYSIN      DD *
   CREATE: PROC OPTIONS(MAIN);

      DCL
        FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
        IN FILE RECORD INPUT,
        STRING CHAR(80),
        EOF BIT(1) INIT('0'B);

      ON ENDFILE(IN) EOF='1'B;

      READ FILE(IN) INTO (STRING);
      DO I=1 BY 1 WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
        WRITE FILE(FAMFILE) FROM (STRING);
        READ FILE(IN) INTO (STRING);
      END;

      PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
    END;
/*
//LKED.SYSLMOD  DD  DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
//        UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.FAMFILE DD   DSNAME=PLIVSAM.AJC1.BASE,DISP=OLD
//GO.IN      DD     *
FRED                     69         M
ANDY                     70         M
SUZAN                    72         F
/*
```

Figure 41. Defining and loading an entry-sequenced data set (ESDS)

Updating an ESDS

Figure 42 shows the addition of a new record on the end of an ESDS. This is done by executing again the program shown in Figure 41. A SEQUENTIAL OUTPUT file is used and the data set associated with it by use of the DSNAME parameter specifying the name PLIVSAM.AJC1.BASE specified in the DEFINE command shown in Figure 41.

```
//OPT9#8   JOB
//STEP1    EXEC  PGM=PGMA
//STEPLIB   DD   DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP)
//          DD   DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT  DD   SYSOUT=A
//FAMFILE   DD   DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//IN        DD   *
JANE                     75         F
//
```

Figure 42. Updating an ESDS

You can rewrite existing records in an ESDS, provided that the length of the record is not changed. You can use a SEQUENTIAL or KEYED SEQUENTIAL update file to do this. If you use keys, they must be RBAs.

Delete is not allowed for ESDS.

Key-sequenced and indexed entry-sequenced data sets

The statements and options allowed for indexed VSAM data sets are shown in Table 27. An indexed data set must be a KSDS with its prime index. Except where otherwise stated, the following description applies to all indexed VSAM data sets.

Table 27 (Page 1 of 2). Statements and options allowed for loading and accessing VSAM indexed data sets

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

Table 27 (Page 2 of 2). Statements and options allowed for loading and accessing VSAM indexed data sets

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

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the declaration.

   The UNLOCK statement for DIRECT UPDATE files is ignored if you use it for files associated with a VSAM KSDS.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);

Loading a KSDS or indexed ESDS

When a KSDS is being loaded, you must open the associated file for KEYED SEQUENTIAL OUTPUT. You must present the records in ascending key order, and you must use the KEYFROM option. Note that you must use the prime index for loading the data set.

If a KSDS already contains some records, and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can only add records at the end of the data set. The rules given in the previous paragraph apply; in particular, the first record you present must have a key greater than the highest key present on the data set.

Figure 43 on page 191 shows the DEFINE command used to define a KSDS. The data set is given the name PLIVSAM.AJC2.BASE and defined as a KSDS because of the use of the INDEXED operand. The position of the keys within the record is defined in the KEYS operand.

Within the PL/I program, a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A KSDS must be loaded in this manner.

The file is associated with the data set by a DD statement which uses the name given in the DEFINE command as the DSNAME parameter.

```
//OPT9#12  JOB
//   EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT  DD  SYSOUT=A
//SYSIN     DD  *
   DEFINE CLUSTER -
     (NAME(PLIVSAM.AJC2.BASE) -
     VOLUMES(nnnnnn) -
     INDEXED -
     TRACKS(3 1) -
     KEYS(20 0) -
     RECORDSIZE(23 80))
/*
//   EXEC IBMZCBG
//PLI.SYSIN      DD *
 TELNOS: PROC OPTIONS(MAIN);

        DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD POS(1),
            NUMBER CHAR(3) DEF CARD POS(21),
            OUTREC CHAR(23) DEF CARD POS(1),
            EOF BIT(1) INIT('0'B);

        ON ENDFILE(SYSIN) EOF='1'B;

        OPEN FILE(DIREC) OUTPUT;

        GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
        WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;

       CLOSE FILE(DIREC);

        END TELNOS;
/*
//GO.DIREC  DD  DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN  DD  *
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,D.        141
CORY,G.             336
ELLIOTT,D.          875
FIGGINS,S.          413
HARVEY,C.D.W.       205
HASTINGS,G.M.       391
KENDALL,J.G.        294
LANCASTER,W.R.      624
MILES,R.            233
NEWMAN,M.W.         450
PITT,W.H.           515
ROLF,D.E.           114
SHEERS,C.D.         241
SUTCLIFFE,M.        472
TAYLOR,G.C.         407
WILTON,L.W.         404
WINSTONE,E.M.       307
//
```

Figure 43. Defining and loading a key-sequenced data set (KSDS)

Using a SEQUENTIAL file to access a KSDS or indexed ESDS

You can open a SEQUENTIAL file that is used to access a KSDS with either the INPUT or the UPDATE attribute.

For READ statements without the KEY option, the records are recovered in ascending key order (or in descending key order if the BKWD option is used). You can obtain the key of a record recovered in this way by means of the KEYTO option.

If you use the KEY option, the record recovered by a READ statement is the one with the specified key. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, without respect to the position of any previous access. If you are accessing the data set via a unique index, the KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists on the data set. For a nonunique index, subsequent retrieval of records with the same key is in the order that they were added to the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the first record with the specified key; otherwise, it is the record that was accessed by the previous READ statement.

Using a DIRECT file to access a KSDS or indexed ESDS

You can open a DIRECT file that is used to access an indexed VSAM data set with the INPUT, OUTPUT, or UPDATE attribute. Do not use a DIRECT file to access the data set via a nonunique index.

If you use a DIRECT OUTPUT file to add records to the data set, and if an attempt is made to insert a record with the same key as a record that already exists, the KEY condition is raised.

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

Figure 44 on page 193 shows one method by which a KSDS can be updated using the prime index.

```
//OPT9#13  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 DIRUPDT: PROC OPTIONS(MAIN);

         DCL DIREC FILE RECORD KEYED ENV(VSAM),
             ONCODE BUILTIN,
             OUTREC CHAR(23),
             NUMBER CHAR(3) DEF OUTREC POS(21),
             NAME CHAR(20) DEF OUTREC,
             CODE CHAR(1),
             EOF BIT(1) INIT('0'B);

         ON ENDFILE(SYSIN) EOF='1'B;

         ON KEY(DIREC) BEGIN;
          IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                           ('NOT FOUND: ',NAME)(A(15),A);
          IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                          ('DUPLICATE: ',NAME)(A(15),A);
          END;

         OPEN FILE(DIREC) DIRECT UPDATE;

       GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
         (COLUMN(1),A(20),A(3),A(1));
       DO WHILE (¬EOF);
       PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
       (A(1),A(20),A(1),A(3),A(1),A(1));
       SELECT (CODE);
          WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
          WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
          WHEN('D') DELETE FILE(DIREC) KEY(NAME);
          OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME) (A(15),A);
       END;
       GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
         (COLUMN(1),A(20),A(3),A(1));
       END;
```

Figure 44 (Part 1 of 2). Updating a KSDS

```
          CLOSE FILE(DIREC);
          PUT FILE(SYSPRINT) PAGE;
          OPEN FILE(DIREC) SEQUENTIAL INPUT;

          EOF='0'B;
          ON ENDFILE(DIREC) EOF='1'B;

          READ FILE(DIREC) INTO(OUTREC);
          DO WHILE(¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
          READ FILE(DIREC) INTO(OUTREC);
          END;
          CLOSE FILE(DIREC);
    END DIRUPDT;
/*
//GO.DIREC DD DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.        516C
GOODFELLOW,D.T.    889A
MILES,R.              D
HARVEY,C.D.W.      209A
BARTLETT,S.G.      183A
CORY,G.               D
READ,K.M.          001A
PITT,W.H.
ROLF,D.F.             D
ELLIOTT,D.         291C
HASTINGS,G.M.         D
BRAMLEY,O.H.       439C
/*
```

Figure 44 (Part 2 of 2). Updating a KSDS

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

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

At the label NEXT, the name, number, and code are read in and action taken according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished (at the label PRINT), the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

The file is associated with the data set by a DD statement that uses the DSNAME PLIVSAM.AJC2.BASE defined in the Access Method Services DEFINE CLUSTER command in Figure 43 on page 191.

Methods of updating a KSDS: There are a number of methods of updating a KSDS. The method shown using a DIRECT file is suitable for the data as it is shown in the example. For mass sequential insertion, use a KEYED SEQUENTIAL UPDATE file. This gives faster performance because the data is written onto the data set only when strictly necessary and not after every write statement, and because the balance of free space within the data set is retained.

Statements to achieve effective mass sequential insertion are:

```
DCL DIREC KEYED SEQUENTIAL UPDATE
    ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
 KEYFROM(NAME);
```

The PL/I input/output routines detect that the keys are in sequence and make the correct requests to VSAM. If the keys are not in sequence, this too is detected and no error occurs, although the performance advantage is lost.

Relative-record data sets

The statements and options allowed for VSAM relative-record data sets (RRDS) are shown in Table 28.

Table 28 (Page 1 of 2). Statements and options allowed for loading and accessing VSAM relative-record data sets

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

Table 28 (Page 2 of 2). Statements and options allowed for loading and accessing VSAM relative-record data sets

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

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, your declaration must also include the attribute KEYED.

   The UNLOCK statement for DIRECT UPDATE files is ignored if you use it for files associated with a VSAM RRDS.

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

Loading an RRDS

When an RRDS is being loaded, you must open the associated file for OUTPUT. Use either a DIRECT or a SEQUENTIAL file.

For a DIRECT OUTPUT file, each record is placed in the position specified by the relative record number (or key) in the KEYFROM option of the WRITE statement (see “Keys for VSAM data sets” on page 181).

For a SEQUENTIAL OUTPUT file, use WRITE statements with or without the KEYFROM option. If you specify the KEYFROM option, the record is placed in the specified slot; if you omit it, the record is placed in the slot following the current position. There is no requirement for the records to be presented in ascending relative record number order. If you omit the KEYFROM option, you can obtain the relative record number of the written record by means of the KEYTO option.

If you want to load an RRDS sequentially, without use of the KEYFROM or KEYTO options, your file is not required to have the KEYED attribute.

It is an error to attempt to load a record into a position that already contains a record: if you use the KEYFROM option, the KEY condition is raised; if you omit it, the ERROR condition is raised.

In Figure 45 on page 197, the data set is defined with a DEFINE CLUSTER command and given the name PLIVSAM.AJC3.BASE. The fact that it is an RRDS is determined by the NUMBERED keyword. In the PL/I program, it is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

If the data had been in order and the keys in sequence, it would have been possible to use a SEQUENTIAL file and write into the data set from the start. The records would then have been placed in the next available slot and given the appropriate number. The number of the key for each record could have been returned using the KEYTO option.

The PL/I file is associated with the data set by the DD statement, which uses as the DSNAME the name given in the DEFINE CLUSTER command.

```
//OPT9#17  JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD *
         DEFINE CLUSTER -
             (NAME(PLIVSAM.AJC3.BASE) -
             VOLUMES(nnnnnn) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20))
/*
//STEP2  EXEC IBMZCBG
//PLI.SYSIN      DD *
 CRR1:   PROC OPTIONS(MAIN);
         DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
             CARD CHAR(80),
             NAME CHAR(20) DEF CARD,
             NUMBER CHAR(2) DEF CARD POS(21),
             IOFIELD CHAR(20),
             EOF BIT(1) INIT('0'B);
         ON ENDFILE (SYSIN) EOF='1'B;
         OPEN FILE(NOS);
         GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         IOFIELD=NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         GET FILE(SYSIN) EDIT(CARD)(A(80));
         END;
         CLOSE FILE(NOS);
 END CRR1;
```

Figure 45 (Part 1 of 2). Defining and loading a relative-record data set (RRDS)

```
/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.       28
CHEESNAME,L.       11
CORY,G.            36
ELLIOTT,D.         85
FIGGINS.E.S.       43
HARVEY,C.D.W.      25
HASTINGS,G.M.      31
KENDALL,J.G.       24
LANCASTER,W.R.     64
MILES,R.           23
NEWMAN,M.W.        40
PITT,W.H.          55
ROLF,D.E.          14
SHEERS,C.D.        21
SURCLIFFE,M.       42
TAYLOR,G.C.        47
WILTON,L.W.        44
WINSTONE,E.M.      37
//
```

Figure 45 (Part 2 of 2). Defining and loading a relative-record data set (RRDS)

Using a SEQUENTIAL file to access an RRDS

You can open a SEQUENTIAL file that is used to access an RRDS with either the INPUT or the UPDATE attribute. If you use any of the options KEY, KEYTO, or KEYFROM, your file must also have the KEYED attribute.

For READ statements without the KEY option, the records are recovered in ascending relative record number order. Any empty slots in the data set are skipped.

If you use the KEY option, the record recovered by a READ statement is the one with the relative record number you specify. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with or without the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, regardless of the position of any previous access. For WRITE with the KEYFROM option, the KEY condition is raised if an attempt is made to insert a record with the same relative record number as a record that already exists on the data set. If you omit the KEYFROM option, an attempt is made to write the record in the next slot, relative to the current position. The ERROR condition is raised if this slot is not empty.

You can use the KEYTO option to recover the key of a record that is added by means of a WRITE statement without the KEYFROM option.

REWRITE statements, with or without the KEY option, are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the record with the relative record number you specify; otherwise, it is the record that was accessed by the previous READ statement.

DELETE statements, with or without the KEY option, can be used to delete records from the dataset.

Using a DIRECT file to access an RRDS

A DIRECT file used to access an RRDS can have the OUTPUT, INPUT, or UPDATE attribute. You can read, write, rewrite, or delete records exactly as though a KEYED SEQUENTIAL file were used.

Figure 46 on page 200 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under VSAM.

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

The PL/I file is associated with the data sets by a DD statement that specifies the DSNAME PLIVSAM.AJC3.BASE, the name given in the DEFINE CLUSTER command in Figure 46 on page 200.

At the end of the example, the DELETE command is used to delete the data set.

```
//* NOTE: WITH A WRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A “DUPLICATE” MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO CORRESPONDS TO AN EXISTING NUMBER IN THE LIST,
//*       FOR EXAMPLE, ELLIOT.
//*       WITH A REWRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
//*       A “NOT FOUND” MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
//*       WHOSE NEWNO DOES NOT CORRESPOND TO AN EXISTING NUMBER IN
//*       THE LIST, FOR EXAMPLE, NEWMAN AND BRAMLEY.
//OPT9#18  JOB
//STEP1  EXEC IBMZCBG
//PLI.SYSIN      DD *
 ACR1:  PROC OPTIONS(MAIN);
                DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
                  (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
                  BYTE CHAR(1) DEF IOFIELD, EOF BIT(1) INIT('0'B),
                  ONCODE BUILTIN;
        ON ENDFILE(SYSIN) EOF='1'B;
        OPEN FILE(NOS) DIRECT UPDATE;
        ON KEY(NOS) BEGIN;
           IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                    ('NOT FOUND:',NAME)(A(15),A);
           IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                    ('DUPLICATE:',NAME)(A(15),A);
        END;
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        DO WHILE (¬EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
           (A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
        SELECT(CODE);
           WHEN('A') WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           WHEN('C') DO;
            DELETE FILE(NOS) KEY(OLDNO);
            WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
           END;
           WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
           OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME)(A(15),A);
        END;
```

Figure 46 (Part 1 of 2). Updating an RRDS

```
        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        END;
         CLOSE FILE(NOS);
            PRINT:
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(NOS) SEQUENTIAL INPUT;
         EOF='0'B;
         ON ENDFILE(NOS) EOF='1'B;
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          DO WHILE (¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD)(A(5),A);
          READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          END;
       CLOSE FILE(NOS);
 END ACR1;
/*
//GO.NOS     DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN   DD  *
NEWMAN,M.W.        5640C
GOODFELLOW,D.T.    89  A
MILES,R.             23D
HARVEY,C.D.W.      29  A
BARTLETT,S.G.      13  A
CORY,G.              36D
READ,K.M.          01  A
PITT,W.H.            55
ROLF,D.F.            14D
ELLIOTT,D.         4285C
HASTINGS,G.M.        31D
BRAMLEY,O.H.       4928C
//STEP3    EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN   DD   *
      DELETE -
          PLIVSAM.AJC3.BASE
//
```

Figure 46 (Part 2 of 2). Updating an RRDS

Part 4. Improving your program

Chapter 11. Improving performance

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

Selecting compile-time options for optimal performance

The compile-time options you choose can greatly improve the performance of the code generated by the compiler; however, like most performance considerations, there are trade-offs associated with these choices. Fortunately, you can weigh the trade-offs associated with compile-time options without editing your source code because these options can be specified on the command line or in the configuration file.

If you want to avoid details, the least complex way to improve the performance of generated code is to specify the following (nondefault) compile-time options:

```
OPT(2)
DFT(REORDER)
```

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 occurs, for instance, when a branch table is created for a SELECT statement where the values in the WHEN clauses contain gaps. The increased number of instructions generated is usually offset by the execution of fewer instructions in other places.

GONUMBER

Using this option results in a statement number table used for debugging. This added information can be extremely helpful when debugging, but including statement number tables increases the size of your executable file. Larger executable files can take longer to load.

RULES

When you use the RULES(IBM) option, the compiler supports scaled FIXED BINARY and, what is more important for performance, generates scaled FIXED BINARY results in some operations. Under RULES(ANS), scaled FIXED BINARY is not supported and scaled FIXED BINARY results are never generated. This means that the code generated under RULES(ANS) always runs at least as fast as the code generated under RULES(IBM), and sometimes runs faster.

For example, consider the following code fragment:

```
dcl (i,j,k) fixed bin(15);
   ⋮
i = j / k;
```

Under RULES(IBM), the result of the division has the attributes FIXED BIN(31,16). This means that a shift instruction is required before the division and several more instructions are needed to perform the assignment.

Under RULES(ANS), the result of the division has the attributes FIXED BIN(15,0). This means that a shift is not needed before the division, and no extra instructions are needed to perform the assignment.

PREFIX

This option determines if selected PL/I conditions are enabled by default. The default suboptions for PREFIX are set to conform to the PL/I language definition; however, overriding the defaults can have a significant effect on the performance of your program. The default suboptions are:

CONVERSION
INVALIDOP
FIXEDOVERFLOW
OVERFLOW
INVALIDOP
NOSIZE
NOSTRINGRANGE
NOSTRINGSIZE
NOSUBSCRIPTRANGE
UNDERFLOW
ZERODIVIDE

By specifying the SIZE, STRINGRANGE, STRINGSIZE, or SUBSCRIPTRANGE suboptions, the compiler generates extra code that helps you pinpoint various problem areas in your source that would otherwise be hard to find. This extra code, however, can slow program performance significantly.

CONVERSION

When you disable the CONVERSION condition, some character-to-numeric conversions are done inline and without checking the validity of the source; therefore, specifying NOCONVERSION also affects program performance.

FIXEDOVERFLOW

On some platforms, the FIXEDOVERFLOW condition is raised by the hardware and the compiler does not need to generate any extra code to detect it.

DEFAULT

Using the DEFAULT option, you can select attribute defaults. As is true with the PREFIX option, the suboptions for DEFAULT are set to conform to the PL/I language definition. Changing the defaults in some instances can affect performance. The default suboptions are:

IBM
BYADDR
RETURNS(BYADDR)

NONCONNECTED
DESCRIPTOR
ORDER
DESCLOCATOR
ASSIGNABLE LINKAGE(OPTLINK)
EBCDIC
NATIVE
NOINLINE

The IBM/ANS and ASSIGNABLE/NONASSIGNABLE 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. Passing a variable by reference, however, can hinder performance in two ways:

1. Every reference to that parameter requires an extra instruction.
2. Since the address of the variable is passed to another routine, the compiler is forced to make assumptions about when that variable might change and generate very conservative code for any reference to that variable.

Consequently, you should pass parameters by value using the BYVALUE suboption whenever your program logic allows. Even if you use the BYADDR attribute to indicate that one parameter should be passed by reference, you can use the DEFAULT(BYVALUE) option to ensure that all other parameters are passed by value.

If a procedure receives and modifies only one parameter that is passed by BYADDR, consider converting the procedure to a function that receives that parameter by value. The function would then end with a RETURN statement containing the updated value of the parameter.

Procedure with BYADDR parameter

```
a: proc( parm1, parm2, ..., parmN );

  dcl parm1 byaddr  ...;
  dcl parm2 byvalue ...;
       ⋮
  dcl parmN byvalue ...;

  /* program logic */

end;
```

Faster, equivalent function with BYVALUE parameter

```
a: proc( parm1, parm2, ..., parmN )
   returns( ... /* attributes of parm1 */ );

  dcl parm1 byvalue  ...;
  dcl parm2 byvalue ...;
      ⋮
  dcl parmN byvalue ...;

  /* program logic */

  return( parm1 );

end;
```

(NON)CONNECTED

The DEFAULT(NONCONNECTED) option indicates that the compiler assumes that any aggregate parameters are NONCONNECTED. References to elements of NONCONNECTED aggregate parameters require the compiler to generate code to access the parameter's descriptor, even if the aggregate is declared with constant extents.

The compiler does not generate these instructions if the aggregate parameter has constant extents and is CONNECTED. Consequently, if your application never passes nonconnected parameters, your code is more optimal if you use the DEFAULT(CONNECTED) option.

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 optimization 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.

(NO)INLINE

The suboption NOINLINE indicates that procedures and begin blocks should not be inlined.

Inlining occurs only when you specify optimization.

Inlining user code eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer, resulting in faster code performance. Inlining produces the best results when the overhead for the function is nontrivial, for example, when functions are called within nested loops. Inlining is also beneficial when the inlined function provides additional opportunities for optimization, such as when constant arguments are used.

For programs containing many procedures that are not nested:

- If the procedures are small and only called from a few places, you can increase performance by specifying INLINE.
- If the procedures are large and called from several places, inlining duplicates code throughout the program. This increase in the size of the program might offset any increase of speed. In this case, you might prefer to leave NOINLINE as the default and specify OPTIONS(INLINE) only on individually selected procedures.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends. Ensure that you have enough stack space for the local storage of the inlined functions.

Summary of compile-time options that improve performance

In summary, the following options (if appropriate for your application) can improve performance:

```
OPTIMIZE(2)
IMPRECISE
RULES(ANS)
DEFAULT with the following suboptions
    (BYVALUE
    RETURNS(BYVALUE)
    CONNECTED
```

NODESCRIPTOR
REORDER
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 `getenv` is an input-only CHAR VARYINGZ string:

```
dcl getenv     entry( char(*) varyingz nonasgn byaddr,
                      pointer byaddr )
               returns( native fixed bin(31) optional )
               options( nodescriptor );
```

If this function is invoked with the string 'IBM_OPTIONS', the compiler can pass the address of that string rather than assigning it to a compiler-generated temporary storage area and passing the address of that area.

GOTO statements

A GOTO statement that uses either a label in another block or a label variable severely limits optimizations that the compiler might perform. If a label array is initialized and declared AUTOMATIC, either implicitly or explicitly, any GOTO to an element of that array will hinder optimization. However, if the array is declared as STATIC, the compiler assumes the CONSTANT attribute for it and no optimization is hindered.

String assignments

When one string is assigned to another, the compiler ensures that:

- The target has the correct value even if the source and target overlap.
- The source string is truncated if it is longer than the target.

This assurance comes at the price of some extra instructions. The compiler attempts to generate these extra instructions only when necessary, but often you, as the programmer, know they are not necessary when the compiler cannot be sure. For instance, if the source and target are based character strings and you know they cannot overlap, you could use the PLIMOVE built-in function to eliminate the extra code the compiler would otherwise be forced to generate.

In the example which follows, faster code is generated for the second assignment statement:

```
dcl based_Str   char(64) based( null() );
dcl target_Addr pointer;
dcl source_Addr pointer;

target_Addr->based_Str = source_Addr->based_Str;

call plimove( target_Addr, source_Addr, stg(based_Str) );
```

If you have any doubts about whether the source and target might overlap or whether the target is big enough to hold the source, you should not use the PLIMOVE built-in.

Loop control variables

Program performance improves if your loop control variables are one of the types in the following list. You should rarely, if ever, use other types of variables.

FIXED BINARY with zero scale factor
FLOAT
ORDINAL
HANDLE
POINTER
OFFSET

Performance also improves if loop control variables are not members of arrays, structures, or unions. The compiler issues a warning message when they are. Loop control variables that are AUTOMATIC and not used for any other purpose give you the optimal code generation.

If a loop control variable is a FIXED BIN, performance is best if it has precision 31 and is SIGNED.

Performance is decreased if your program depends not only on the value of a loop control variable, but also on its address. For example, if the ADDR built-in function is applied to the variable or if the variable is passed BYADDR to another routine.

PACKAGEs versus nested PROCEDUREs

Calling nested procedures requires that an extra *hidden parameter* (the backchain pointer) is passed. As a result, the fewer nested procedures that your application contains, the faster it runs.

To improve the performance of your application, you can convert a mother-daughter pair of nested procedures into level-1 sister procedures inside of a package. This conversion is possible if your nested procedure does not rely on any of the automatic and internal static variables declared in its parent procedures.

If procedure b in “Example with nested procedures” does not use any of the variables declared in a, you can improve the performance of both procedures by reorganizing them into the package illustrated in “Example with packaged procedures.”

Example with nested procedures

```
a: proc;

  dcl (i,j,k) fixed bin;
  dcl ib      based fixed bin;
         ⋮
  call b( addr(i) );
         ⋮
  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;
end;
```

Example with packaged procedures

```
p: package exports( a );

  dcl ib      based fixed bin;

  a: proc;

    dcl (i,j,k) fixed bin;
         ⋮
    call b( addr(i) );
         ⋮
  end;

  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;

end p;
```

REDUCIBLE Functions

REDUCIBLE indicates that a procedure or entry need not be invoked multiple times if the argument(s) stays unchanged, and that the invocation of the procedure has no side effects.

For example, a user-written function that computes a result based on unchanging data should be declared REDUCIBLE. A function that computes a result based on changing data, such as a random number or time of day, should be declared IRREDUCIBLE.

In the following example, *f* is invoked only once since REDUCIBLE is part of the declaration. If IRREDUCIBLE had been used in the declaration, *f* would be invoked twice.

```
dcl (f) entry options( reducible ) returns( fixed bin );

select;
  when( f(x) < 0 )
      ⋮
  when( f(x) > 0 )
      ⋮
  otherwise
      ⋮
end;
```

DESCLOCATOR or DESCLIST

When the DEFAULT(DESCLOCATOR) option is in effect, the compiler passes arguments requiring descriptors (such as strings and structures) via a descriptor locator in much the same way that the old compiler did. More information on descriptors and how they are passed is available in Chapter 13, “PL/I - Language Environment descriptors” on page 238.

This option allows you to invoke an entry point that is not always passed all of the arguments that it declares.

This option also allows you to continue the somewhat unwise programming practice of passing a structure and receiving it as a pointer.

However, the code generated by the compiler for DEFAULT(DESCLOCATOR) may, in some situations, perform less well than that for DEFAULT(DESCLIST).

DEFINED versus UNION

The UNION attribute is more powerful than the DEFINED attribute and provides more function. In addition, the compiler generates better code for union references.

In the following example, the pair of variables `b3` and `b4` perform the same function as `b1` and `b2`, but the compiler generates more optimal code for the pair in the union.

```
dcl b1 bit(31);
dcl b2 bit(16) def b1;

dcl
  1 * union,
    2 b3 bit(32),
    2 b4 bit(16);
```

Code that uses UNIONs instead of the DEFINED attribute is subject to less misinterpretation. Variable declarations in unions are in a single location making it easy to realize that when one member of the union changes, all of the others change also. This dynamic change is less obvious in declarations that use DEFINED variables since the declare statements can be several lines apart.

Named constants versus static variables

You can define named constants by declaring a variable with the VALUE attribute. If you use static variables with the INITIAL attribute and you do not alter the variable, you should declare the variable a named constant using the VALUE attribute. The compiler does not treat NONASSIGNABLE scalar STATIC variables as true named constants.

The compiler generates better code whenever expressions are evaluated during compilation, so you can use named constants to produce efficient code with no loss in readability. For example, identical object code is produced for the two usages of the VERIFY built-in function in the following example:

```
dcl numeric char  value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );
```

The following examples illustrate how you can use the VALUE attribute to get optimal code without sacrificing readability.

Example with optimal code but no meaningful names

```
dcl  x  bit(8) aligned;

select( x );
  when( '01'b4 )
        ⋮
  when( '02'b4 )
        ⋮
  when( '03'b4 )
        ⋮
end;
```

Example with meaningful names but not optimal code

```
dcl (  a1  init( '01'b4)
      ,a2  init( '02'b4)
      ,a3  init( '03'b4)
      ,a4  init( '04'b4)
      ,a5  init( '05'b4)
    ) bit(8) aligned static nonassignable;

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
        ⋮
  when( a2 )
        ⋮
  when( a3 )
        ⋮
end;
```

Example with optimal code AND meaningful names

```
dcl (  a1  value( '01'b4)
      ,a2  value( '02'b4)
      ,a3  value( '03'b4)
      ,a4  value( '04'b4)
      ,a5  value( '05'b4)
    ) bit(8);

dcl  x  bit(8) aligned;

select( x );
  when( a1 )
        ⋮
  when( a2 )
        ⋮
  when( a3 )
        ⋮
end;
```

Avoiding calls to library routines

The bitwise operations (prefix NOT, infix AND, infix OR, and infix EXCLUSIVE OR) are often evaluated by calls to library routines. These operations are, however, handled without a library call if either of the following conditions is true:

- Both operands are bit(1)
- Both operands are aligned and have the same constant length.

For certain assignments, expressions, and built-in function references, the compiler generates calls to library routines. If you avoid these calls, your code generally runs faster.

To help you determine when the compiler generates such calls, the compiler generates a message whenever a conversion is done using a library routine.

Part 5. Using interfaces to other products

Chapter 12. Using the Sort program

The compiler provides an interface called PLISRTx (x = A, B, C, or D) that allows you to make use of the IBM-supplied Sort programs.

To use the Sort program with PLISRTx, you must:

1. Include a call to one of the entry points of PLISRTx, passing it the information on the fields to be sorted. This information includes the length of the records, the maximum amount of storage to use, the name of a variable to be used as a return code, and other information required to carry out the sort.
2. Specify the data sets required by the Sort program in JCL DD statements.

When used from PL/I, the Sort program sorts records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted can be either a data set or a user-written PL/I procedure that the Sort program will call each time a record is required for the sort. Similarly, the destination of the sort can be a data set or a PL/I procedure that handles the sorted records.

Using PL/I procedures allows processing to be done before or after the sort itself, thus allowing a complete sorting operation to be handled completely by a call to the sort interface. It is important to understand that the PL/I procedures handling input or output are called from the Sort program itself and will effectively become part of it.

PL/I can operate with DFSORT™ or a program with the same interface. DFSORT is a release of the program product 5740-SM1. DFSORT has many built-in features you can use to eliminate the need for writing program logic (for example, INCLUDE, OMIT, OUTREC and SUM statement plus the many ICETOOL operators). See *DFSORT Application Programming Guide* for details and *Getting Started with DFSORT* for a tutorial.

The following material applies to DFSORT. Because you can use programs other than DFSORT, the actual capabilities and restrictions vary. For these capabilities and restrictions, see *DFSORT Application Programming Guide*, or the equivalent publication for your sort product.

To use the Sort program you must include the correct PL/I statements in your source program and specify the correct data sets in your JCL.

Preparing to use Sort

Before using Sort, you must determine the type of sort you require, the length and format of the sorting fields in the data, the length of your data records, and the amount of auxiliary and main storage you will allow for sorting.

To determine the PLISRTx entry point that you will use, you must decide the source of your unsorted data, and the destination of your sorted data. You must choose between data sets and PL/I subroutines. Using data sets is simpler to understand and gives faster performance. Using PL/I subroutines gives you more flexibility and more function, enabling you to manipulate or print the data before it is sorted, and to make immediate use of it in its sorted form. If you decide to use an input or output handling subroutine, you will need to read “Data input and output handling routines” on page 227.

The entry points and the source and destination of data are as follows:

Entry point	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine
PLISRTD	Subroutine	Subroutine

Having determined the entry point you are using, you must now determine the following things about your data set:

- The position of the sorting fields; these can be either the complete record or any part or parts of it
- The type of data these fields represent, for example, character or binary
- Whether you want the sort on each field to be in ascending or descending order
- Whether you want equal records to be retained in the order of the input, or whether their order can be altered during sorting

Specify these options on the SORT statement, which is the first argument to PLISRTx. After you have determined these, you must determine two things about the records to be sorted:

- Whether the record format is fixed or varying
- The length of the record, which is the maximum length for varying

Specify these on the RECORD statement, which is the second argument to PLISRTx.

Finally, you must decide on the amount of main and auxiliary storage you will allow for the Sort program. For further details, see “Determining storage needed for Sort” on page 222.

Choosing the type of Sort

If you want to make the best use of the Sort program, you must understand something of how it works. In your PL/I program you specify a sort by using a CALL statement to the sort interface subroutine PLISRTx. This subroutine has four entry points: x=A, B, C, and D. Each specifies a different source for the unsorted data and destination for the data when it has been sorted. For example, a call to PLISRTA specifies that the unsorted data (the input to sort) is on a data set, and that the sorted data (the output from sort) is to be placed on another data set. The CALL PLISRTx statement must contain an argument list giving the Sort program information about the data set to be sorted, the fields on which it is to be sorted, the amount of space available, the name of a variable into which Sort will place a return code indicating the success or failure of the sort, and the name of any output or input handling procedure that can be used.

The sort interface routine builds an argument list for the Sort program from the information supplied by the PLISRTx argument list and the choice of PLISRTx entry point. Control is then transferred to the Sort program. If you have specified an output- or input-handling routine, this will be called by the Sort program as many times as is necessary to handle each of the unsorted or sorted records. When the sort operation is complete, the Sort program returns to the PL/I calling procedure communicating its success or failure in a return code, which is placed in one of the arguments passed to the interface routine. The return code can then be tested in the PL/I routine to discover whether processing should continue. Figure 47 is a simplified flowchart showing this operation.

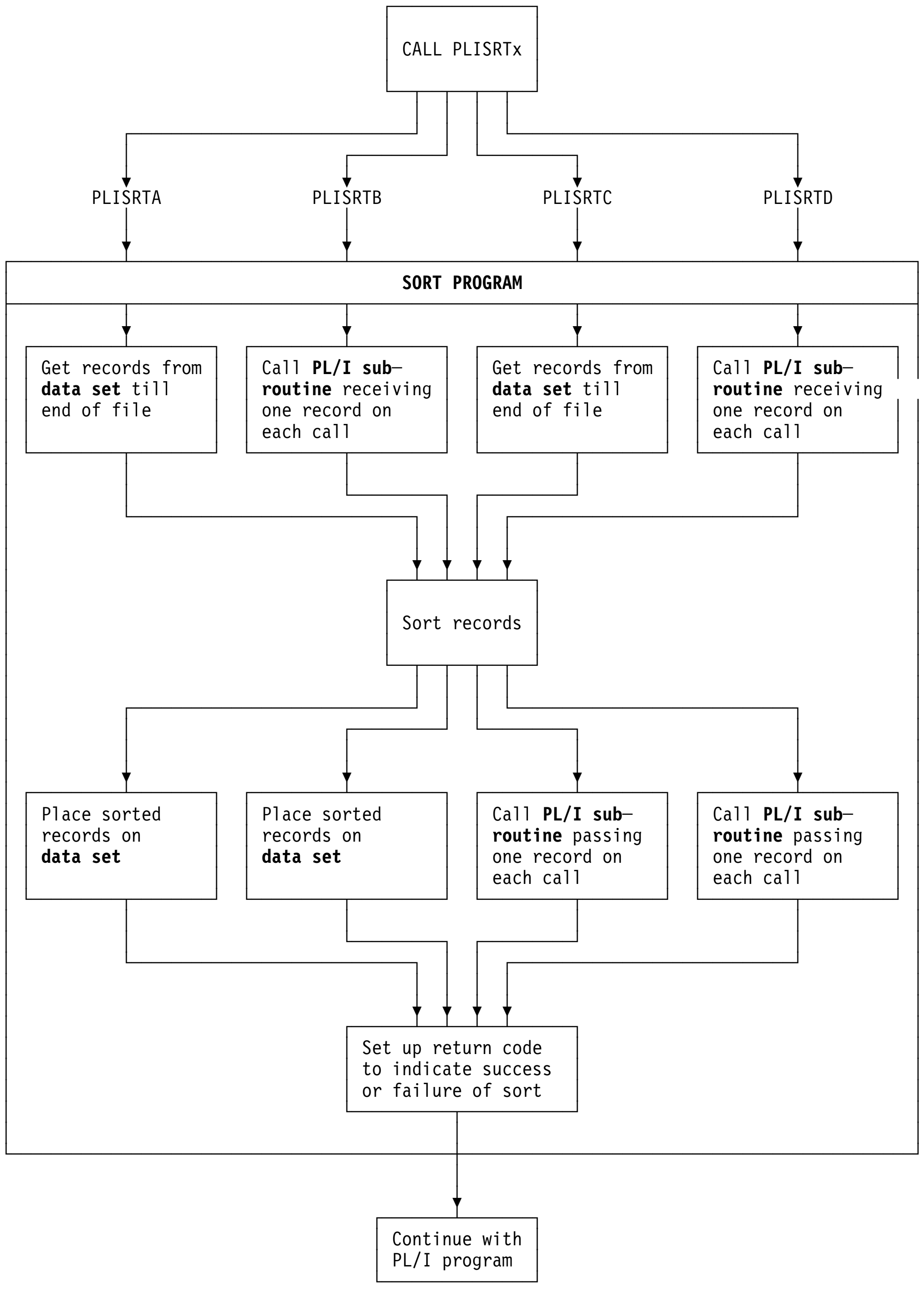

CALL PLISRTx
PLISRTA
PLISRTB
PLISRTC
PLISRTD
SORT PROGRAM
Get records from data set till end of file
Call PL/I sub-routine receiving one record on each call
Get records from data set till end of file
Call PL/I sub-routine receiving one record on each call
Sort records
Place sorted records on data set
Place sorted records on data set
Call PL/I sub-routine passing one record on each call
Call PL/I sub-routine passing one record on each call
Set up return code to indicate success or failure of sort
Continue with PL/I program

Figure 47. Flow of control for Sort program

Within the Sort program itself, the flow of control between the Sort program and input- and output-handling routines is controlled by return codes. The Sort program calls these routines at the appropriate point in its processing. (Within the Sort program, and its associated documentation, these routines are known as *user exits*. The routine that passes input to be sorted is the E15 sort user exit. The routine that processes sorted output is the E35 sort user exit.) From the routines, Sort expects a return code indicating either that it should call the routine again, or that it should continue with the next stage of processing.

The important points to remember about Sort are: (1) it is a self-contained program that handles the complete sort operation, and (2) it communicates with the caller, and with the user exits that it calls, by means of return codes.

The remainder of this chapter gives detailed information on how to use Sort from PL/I. First the required PL/I statements are described, and then the data set requirements. The chapter finishes with a series of examples showing the use of the four entry points of the sort interface routine.

Specifying the sorting field

The SORT statement is the first argument to PLISRTx. The syntax of the SORT statement must be a character string expression that takes the form:

```
'bSORTbFIELDS=(start1,length1,form1,seq1,
...startn,lengthn,formn,seqn)[,other options]b'
```

For example:

```
' SORT FIELDS=(1,10,CH,A) '
```

b represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

start,length,form,seq
defines a sorting field. You can specify any number of sorting fields, but there is a limit on the total length of the fields. If more than one field is to be sorted on, the records are sorted first according to the first field, and then those that are of equal value are sorted according to the second field, and so on. If all the sorting values are equal, the order of equal records will be arbitrary unless you use the EQUALS option. (See later in this definition list.) Fields can overlay each other.

For DFSORT (5740-SM1), the maximum total length of the sorting fields is restricted to 4092 bytes and all sorting fields must be within 4092 bytes of the start of the record. Other sort products might have different restrictions.

start is the starting position within the record. Give the value in bytes except for binary data where you can use a “byte.bit” notation. The first byte in a string is considered to be byte 1, the first bit is bit 0. (Thus the second bit in byte 2 is referred to as 2.1.) For varying length records you must include the 4-byte length prefix, making 5 the first byte of data.

length is the length of the sorting field. Give the value in bytes except for binary where you can use “byte.bit” notation. The length of sorting fields is restricted according to their data type.

form is the format of the data. This is the format assumed for the purpose of sorting. All data passed between PL/I routines and Sort must be in the form of character strings. The main data types and the restrictions on their length are shown below. Additional data types are available for special-purpose sorts. See the *DFSORT Application Programming Guide*, or the equivalent publication for your sort product.

Code	**Data type and length**
CH	character 1–4096
ZD	zoned decimal, signed 1–32
PD	packed decimal, signed 1–32
FI	fixed point, signed 1–256
BI	binary, unsigned 1 bit to 4092 bytes
FL	floating-point, signed 1–256

The sum of the lengths of all fields must not exceed 4092 bytes.

seq is the sequence in which the data will be sorted as follows:

A	ascending (that is, 1,2,3,...)
D	descending (that is, ...,3,2,1)

Note: You cannot specify E, because PL/I does not provide a method of passing a user-supplied sequence.

other options
You can specify a number of other options, depending on your Sort program. You must separate them from the FIELDS operand and from each other by commas. Do not place blanks between operands.

FILSZ=y
specifies the number of records in the sort and allows for optimization by Sort. If y is only approximate, E should precede y.

SKIPREC=y
specifies that y records at the start of the input file are to be ignored before sorting the remaining records.

CKPT or CHKPT
specifies that checkpoints are to be taken. If you use this option, you must provide a SORTCKPT data set and DFSORT's 16NCKPT=NO installation option must be specified.

EQUALS|NOEQUALS
specifies whether the order of equal records will be preserved as it was in the input (EQUALS) or will be arbitrary (NOEQUALS). You could improve sort performance by using the NOEQUALS. The default option is chosen when Sort is installed. The IBM-supplied default is NOEQUALS.

DYNALLOC=(d,n)
(OS/VS Sort only) specifies that the program dynamically allocates intermediate storage.

d	is the device type (3380, etc.).
n	is the number of work areas.

Specifying the records to be sorted

Use the RECORD statement as the second argument to PLISRTx. The syntax of the RECORD statement must be a character string expression which, when evaluated, takes the syntax shown below:

```
'bRECORDbTYPE=rectype[,LENGTH=(L1,[,,L4,L5])]b'
```

For example:

```
' RECORD TYPE=F,LENGTH=(80) '
```

b represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

TYPE
specifies the type of record as follows:

F fixed length
V varying length EBCDIC
D varying length ASCII

Even when you use input and output routines to handle the unsorted and sorted data, you must specify the record type as it applies to the work data sets used by Sort.

If varying length strings are passed to Sort from an input routine (E15 exit), you should normally specify V as a record format. However, if you specify F, the records are padded to the maximum length with blanks.

LENGTH
specifies the length of the record to be sorted. You can omit LENGTH if you use PLISRTA or PLISRTC, because the length will be taken from the input data set. Note that there is a restriction on the maximum and minimum length of the record that can be sorted (see below). For varying length records, you must include the four-byte prefix.

11 is the length of the record to be sorted. For VSAM data sets sorted as varying records it is the maximum record size+4.

,, represent two arguments that are not applicable to Sort when called from PL/I. You must include the commas if the arguments that follow are used.

14 specifies the minimum length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

15 specifies the modal (most common) length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

Maximum record lengths

The length of a record can never exceed the maximum length specified by the user. The maximum record length for variable length records is 32756 bytes and for fixed length records it is 32760 bytes.

Determining storage needed for Sort

Main storage

Sort requires both main and auxiliary storage. The minimum main storage for DFSORT is 88K bytes, but for best performance, more storage (on the order of 1 megabyte or more) is recommended. DFSORT can take advantage of storage above 16M virtual or extended architecture processors. Under OS/390, DFSORT can also take advantage of expanded storage. You can specify that Sort use the maximum amount of storage available by passing a storage parameter in the following manner:

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B||UNSPEC('MAX');
CALL PLISRTA
    (' SORT FIELDS=(1,80,CH,A) ',
     ' RECORD TYPE=F,LENGTH=(80) ',
       MAXSTOR,
       RETCODE,
       'TASK');
```

If files are opened in E15 or E35 exit routines, enough residual storage should be allowed for the files to open successfully.

Auxiliary storage

Calculating the minimum auxiliary storage for a particular sorting operation is a complicated task. To achieve maximum efficiency with auxiliary storage, use direct access storage devices (DASDs) whenever possible. For more information on improving program efficiency, see the *DFSORT Application Programming Guide*, particularly the information about dynamic allocation of workspace which allows DFSORT to determine the auxiliary storage needed and allocate it for you.

If you are interested only in providing enough storage to ensure that the sort will work, make the total size of the SORTWK data sets large enough to hold three sets of the records being sorted. (You will not gain any advantage by specifying more than three if you have enough space in three data sets.)

However, because this suggestion is an approximation, it might not work, so you should check the sort manuals. If this suggestion does work, you will probably have wasted space.

Calling the Sort program

When you have determined the points described above, you are in a position to write the CALL PLISRTx statement. You should do this with some care; for the entry points and arguments to use, see Table 29.

Table 29 (Page 1 of 2). The entry points and arguments to PLISRTx (x = A, B, C, or D)

Entry points	Arguments
PLISRTA Sort input: data set Sort output: data set	(sort statement,record statement,storage,return code [,data set prefix,message level, sort technique])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(sort statement,record statement,storage,return code,input routine [,data set prefix,message level,sort technique])

Table 29 (Page 2 of 2). The entry points and arguments to PLISRTx (x = A, B, C, or D)

Entry points	**Arguments**
PLISRTC Sort input: data set Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,output routine [,data set prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,input routine,output routine[,data set prefix,message level,sort technique])
Sort statement	Character string expression containing the Sort program SORT statement. Describes sorting fields and format. See “Specifying the sorting field” on page 219.
Record statement	Character string expression containing the Sort program RECORD statement. Describes the length and record format of data. See “Specifying the records to be sorted” on page 221.
Storage	Fixed binary expression giving maximum amount of main storage to be used by the Sort program. Must be >88K bytes for DFSORT. See also “Determining storage needed for Sort.”
Return code	Fixed binary variable of precision (31,0) in which Sort places a return code when it has completed. The meaning of the return code is: 0=Sort successful 16=Sort failed 20=Sort message data set missing
Input routine	(PLISRTB and PLISRTD only.) Name of the PL/I external or internal procedure used to supply the records for the Sort program at sort exit E15.
Output routine	(PLISRTC and PLISRTD only.) Name of the PL/I external or internal procedure to which Sort passes the sorted records at sort exit E35.
Data set prefix	Character string expression of four characters that replaces the default prefix of 'SORT' in the names of the sort data sets SORTIN, SORTOUT, SORTWKnn and SORTCNTL, if used. Thus if the argument is “TASK”, the data sets TASKIN, TASKOUT, TASKWKnn, and TASKCNTL can be used. This facility enables multiple invocations of Sort to be made in the same job step. The four characters must start with an alphabetic character and must not be one of the reserved names PEER, BALN, CRCX, OSCL, POLY, DIAG, SYSC, or LIST. You must code a null string for this argument if you require either of the following arguments but do not require this argument.
Message level	Character string expression of two characters indicating how Sort's diagnostic messages are to be handled, as follows: NO No messages to SYSOUT AP All messages to SYSOUT CP Critical messages to SYSOUT SYSOUT will normally be allocated to the printer, hence the use of the mnemonic letter “P”. Other codes are also allowed for certain of the Sort programs. For further details on these codes, see *DFSORT Application Programming Guide*. You must code a null string for this argument if you require the following argument but you do not require this argument.
Sort technique	(This is not used by DFSORT; it appears for compatibility reasons only.) Character string of length 4 that indicates the type of sort to be carried out, as follows: PEER Peerage sort BALN Balanced CRCX Criss-cross sort OSCL Oscillating POLY Polyphase sort Normally the Sort program will analyze the amount of space available and choose the most effective technique without any action from you. You should use this argument only as a bypass for sorting problems or when you are certain that performance could be improved by another technique. See *DFSORT Application Programming Guide* for further information.

The examples below indicate the form that the CALL PLISRTx statement normally takes.

Example 1

A call to PLISRTA sorting 80-byte records from SORTIN to SORTOUT using 1048576 (1 megabyte) of storage, and a return code, RETCODE, declared as FIXED BINARY (31,0).

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
             ' RECORD TYPE=F,LENGTH=(80) ',
               1048576,
               RETCODE);
```

Example 2

This example is the same as example 1 except that the input, output, and work data sets are called TASKIN, TASKOUT, TASKWK01, and so forth. This might occur if Sort was being called twice in one job step.

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
             ' RECORD TYPE=F,LENGTH=(80) ',
               1048576,
               RETCODE,
               'TASK');
```

Example 3

This example is the same as example 1 except that the sort is to be undertaken on two fields. First, bytes 1 to 10 which are characters, and then, if these are equal, bytes 11 and 12 which contain a binary field, both fields are to be sorted in ascending order.

```
CALL PLISRTA (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
             ' RECORD TYPE=F,LENGTH=(80) ',
               1048576,
               RETCODE);
```

Example 4

This example shows a call to PLISRTB. The input is to be passed to Sort by the PL/I routine PUTIN, the sort is to be carried out on characters 1 to 10 of an 80 byte fixed length record. Other information as above.

```
CALL PLISRTB (' SORT FIELDS=(1,10,CH,A) ',
             ' RECORD TYPE=F,LENGTH=(80) ',
               1048576,
               RETCODE,
               PUTIN);
```

Example 5

This example shows a call to PLISRTD. The input is to be supplied by the PL/I routine PUTIN and the output is to be passed to the PL/I routine PUTOUT. The record to be sorted is 84 bytes varying (including the length prefix). It is to be sorted on bytes 1 through 5 of the data in ascending order, then if these fields are equal, on bytes 6 through 10 in descending order. (Note that the 4-byte length prefix is included so that the actual values used are 5 and 10 for the starting points.) If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

```
CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(84) ',
                1048576,
                RETCODE,
                PUTIN,       /*input routine (sort exit E15)*/
                PUTOUT);     /*output routine (sort exit E35)*/
```

Determining whether the Sort was successful

When the sort is completed, Sort sets a return code in the variable named in the fourth argument of the call to PLISRTx. It then returns control to the statement that follows the CALL PLISRTx statement. The value returned indicates the success or failure of the sort as follows:

0	Sort successful
16	Sort failed
20	Sort message data set missing

You must declare the variable to which the return code is passed as FIXED BINARY (31,0). It is standard practice to test the value of the return code after the CALL PLISRTx statement and take appropriate action according to the success or failure of the operation.

For example (assuming the return code was called RETCODE):

```
IF RETCODE¬=0 THEN DO;
  PUT DATA(RETCODE);
  SIGNAL ERROR;
END;
```

If the job step that follows the sort depends on the success or failure of the sort, you should set the value returned in the Sort program as the return code from the PL/I program. This return code is then available for the following job step. The PL/I return code is set by a call to PLIRETC. You can call PLIRETC with the value returned from Sort thus:

```
CALL PLIRETC(RETCODE);
```

You should not confuse this call to PLIRETC with the calls made in the input and output routines, where a return code is used for passing control information to Sort.

Establishing data sets for Sort

If DFSORT was installed in a library not know to the system, you must specify the DFSORT library in a JOBLIB or STEPLIB DD statement.

When you call Sort, certain sort data sets must not be open. These are:

SYSOUT
A data set (normally the printer) on which messages from the Sort program will be written.

Sort work data sets

SORTWK01–SORTWK32

Note: If you specify more than 32 sort work data sets, DFSORT will only use the first 32.

******WK01–****WK32**
From 1 to 32 working data sets used in the sorting process. These must be direct-access. For a discussion of space required and number of data sets, see "Determining storage needed for Sort" on page 222.

**** represents the four characters that you can specify as the data set prefix argument in calls to PLISRTx. This allows you to use data sets with prefixes other than SORT. They must start with an alphabetic character and must not be the names PEER, BALN, CRCX, OSCL, POLY, SYSC, LIST, or DIAG.

Input data set

SORTIN

******IN**
The input data set used when PLISRTA and PLISRTC are called.

See *****WK01–****WK32* above for a detailed description.

Output data set

SORTOUT

******OUT**
The output data set used when PLISRTA and PLISRTB are called.

See *****WK01–****WK32* above for a detailed description.

Checkpoint data set

SORTCKPT

Data set used to hold checkpoint data, if CKPT or CHKPT option was used in the SORT statement argument and DFSORT's 16NCKPT=NO installation option was specified. For information on this program DD statement, see *DFSORT Application Programming Guide*.

DFSPARM
SORTCNTL

Data set from which additional or changed control statements can be read (optional). For additional information on this program DD statement, see *DFSORT Application Programming Guide*.

See *****WK01–****WK32* above for a detailed description.

Sort data input and output

The source of the data to be sorted is provided either directly from a data set or indirectly by a routine (Sort Exit E15) written by the user. Similarly, the destination of the sorted output is either a data set or a routine (Sort Exit E35) provided by the user.

PLISRTA is the simplest of all of the interfaces because it sorts from data set to data set. An example of a PLISRTA program is in Figure 51 on page 231. Other interfaces require either the input handling routine or the output handling routine, or both.

Data input and output handling routines

The input handling and output handling routines are called by Sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRTx, they behave in the same way as ordinary internal procedures in respect of scope of names. The input and output procedure names must themselves be known in the procedure that makes the call to PLISRTx.

The routines are called individually for each record required by Sort or passed from Sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures will not retain their values between calls. Consequently, items such as counters, which need to be retained from one call to the next, should either be declared as STATIC or be declared in the containing block.

E15—Input handling routine (Sort Exit E15)

Input routines are normally used to process the data in some way before it is sorted. You can use input routines to print the data, as shown in the Figure 52 on page 232 and Figure 54 on page 234, or to generate or manipulate the sorting fields to achieve the correct results.

The input handling routine is used by Sort when a call is made to either PLISRTB or PLISRTD. When Sort requires a record, it calls the input routine which should return a record in character string format, with a return code of 12. This return code means that the record passed is to be included in the sort. Sort continues to call the routine until a return code of 8 is passed. A return code of 8 means that all records have already been passed, and that Sort is not to call the routine again. If a record is returned when the return code is 8, it is ignored by Sort.

The data returned by the routine must be a character string. It can be fixed or varying. If it is varying, you should normally specify V as the record format in the RECORD statement which is the second argument in the call to PLISRTx. However, you can specify F, in which case the string will be padded to its maximum length with blanks. The record is returned with a RETURN statement, and you must specify the RETURNS attribute in the PROCEDURE statement. The return code is set in a call to PLIRETC. A flowchart for a typical input routine is shown in Figure 48 on page 228.

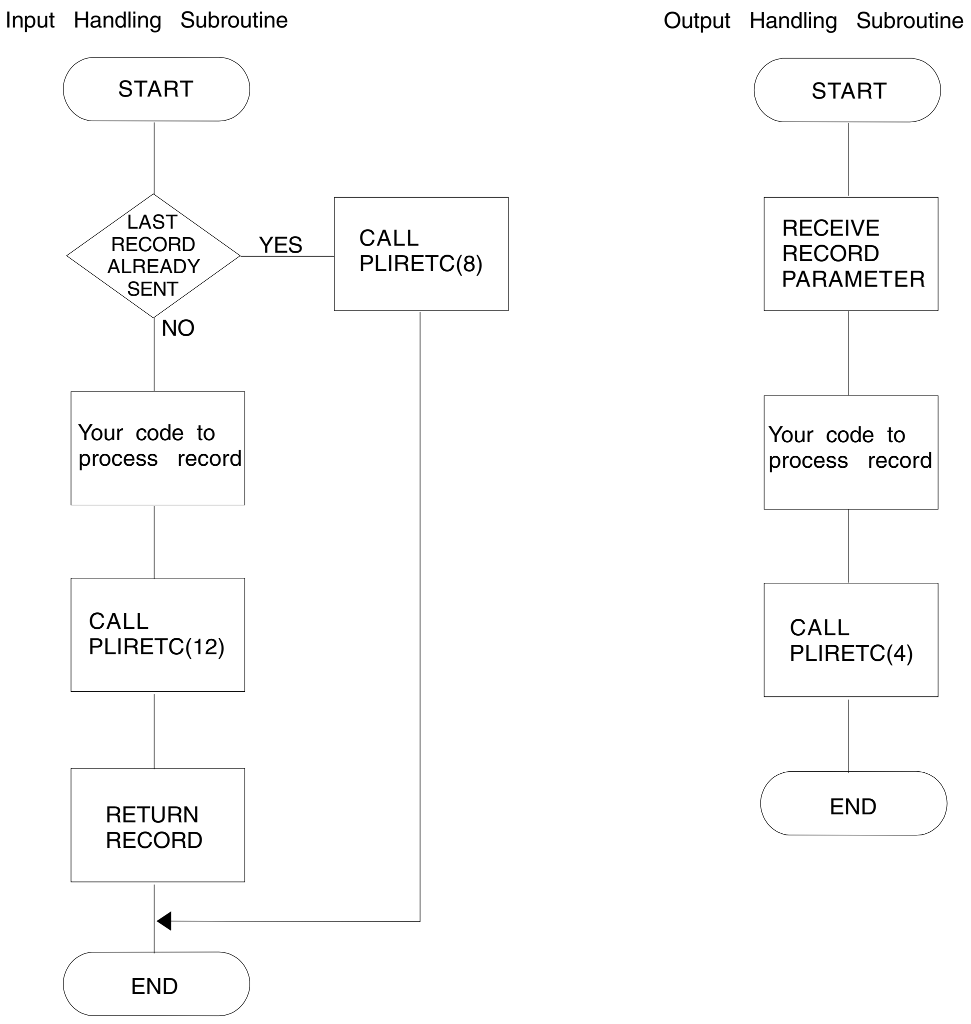


Figure 48. Flowcharts for input and output handling subroutines

Skeletal code for a typical input routine is shown in Figure 49 on page 229.

```
E15: PROC RETURNS (CHAR(80));
     /*---------------------------------------------------------------*/
     /*RETURNS attribute must be used specifying length of data to be */
     /* sorted, maximum length if varying strings are passed to Sort. */
     /*---------------------------------------------------------------*/
   DCL STRING CHAR(80); /*--------------------------------------------*/
                        /*A character string variable will normally be*/
                        /* required to return the data to Sort       */
                        /*--------------------------------------------*/

   IF LAST_RECORD_SENT THEN
     DO;
      /*---------------------------------------------------------------*/
      /*A test must be made to see if all the records have been sent,  */
      /*if they have, a return code of 8 is set up and control returned*/
      /*to Sort                                                        */
      /*---------------------------------------------------------------*/

        CALL PLIRETC(8);  /*--------------------------------------------*/
                          /* Set return code of 8, meaning last record */
                          /* already sent.                             */
                          /*--------------------------------------------*/
        GOTO FINAL;

     END;

   ELSE
     DO;
      /*------------------------------------------------*/
      /* If another record is to be sent to Sort, do the*/
      /* necessary processing, set a return code of 12  */
      /* by calling PLIRETC, and return the data as a   */
      /* character string to Sort                       */
      /*------------------------------------------------*/

   ****(The code to do your processing goes here)

       CALL PLIRETC (12);  /*--------------------------------------*/
                           /* Set return code of 12, meaning this  */
                           /* record is to be included in the sort */
                           /*--------------------------------------*/
       RETURN (STRING);  /*Return data with RETURN statement*/
     END;
FINAL:
  END;   /*End of the input procedure*/
```

Figure 49. Skeletal code for an input procedure

Examples of an input routine are given in Figure 52 on page 232 and Figure 54 on page 234.

In addition to the return codes of 12 (include current record in sort) and 8 (all records sent), Sort allows the use of a return code of 16. This ends the sort and causes Sort to return to your PL/I program with a return code of 16–Sort failed.

Note: A call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When an output handling routine has been used, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRTx to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 53 on page 233.

E35—Output handling routine (Sort Exit E35)

Output handling routines are normally used for any processing that is necessary after the sort. This could be to print the sorted data, as shown in Figure 53 on page 233 and Figure 54 on page 234, or to use the sorted data to generate further information. The output handling routine is used by Sort when a call is made to PLISRTC or PLISRTD. When the records have been sorted, Sort passes them, one at a time, to the output handling routine. The output routine then processes them as required. When all the records have been passed, Sort sets up its return code and returns to the statement after the CALL PLISRTx statement. There is no indication from Sort to the output handling routine that the last record has been reached. Any end-of-data handling must therefore be done in the procedure that calls PLISRTx.

The record is passed from Sort to the output routine as a character string, and you must declare a character string parameter in the output handling subroutine to receive the data. The output handling subroutine must also pass a return code of 4 to Sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

The sort can be stopped by passing a return code of 16 to Sort. This will result in Sort returning to the calling program with a return code of 16–Sort failed.

The record passed to the routine by Sort is a character string parameter. If you specified the record type as F in the second argument in the call to PLISRTx, you should declare the parameter with the length of the record. If you specified the record type as V, you should declare the parameter as adjustable, as in the following example:

```
DCL STRING CHAR(*);
```

Figure 55 on page 235 shows a program that sorts varying length records.

A flowchart for a typical output handling routine is given in Figure 48 on page 228. Skeletal code for a typical output handling routine is shown in Figure 50.

```
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 50. Skeletal code for an output handling procedure

You should note that a call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When you have used an output handling routine, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRTx to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples at the end of this chapter.

Calling PLISRTA example

After each time that the PL/I input- and output-handling routines communicate the return-code information to the Sort program, the return-code field is reset to zero; therefore, it is not used as a regular return code other than its specific use for the Sort program.

For details on handling conditions, especially those that occur during the input- and output-handling routines, see *OS/390 Language Environment Programming Guide*.

```
//OPT14#7  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX106: PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                     1048576
                     RETURN_CODE);
     SELECT (RETURN_CODE);
       WHEN(0) PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT (
            'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
       CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/
       END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*
```

Figure 51. PLISRTA—sorting from input data set to output data set

Calling PLISRTB example

```
//OPT14#8  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX107:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X);
     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER   PUT SKIP EDIT
           ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
     END /* select */;
     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E15X:   /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT
            STREAM AND PUTS THEM BEFORE THEY ARE SORTED*/
     PROC RETURNS (CHAR(80));
          DCL SYSIN FILE RECORD INPUT,
              INFIELD CHAR(80);

          ON ENDFILE(SYSIN) BEGIN;
             PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
             CALL PLIRETC(8);  /* signal that last record has
                                  already been sent to sort*/
             GOTO ENDE15;
             END;

           READ FILE (SYSIN) INTO (INFIELD);
           PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
           CALL PLIRETC(12);  /* request sort to include current
                                 record and return for more*/
          RETURN(INFIELD);
    ENDE15:
          END E15X;
  END EX107;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT  DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 52. PLISRTB—sorting from input handling routine to output data set

Calling PLISRTC example

```
//OPT14#9  JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
 EX108:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E35X);
        SELECT(RETURN_CODE);
          WHEN(0)  PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
          WHEN(16) PUT SKIP EDIT
              ('SORT FAILED, RETURN_CODE 16') (A);
          WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
          OTHER    PUT SKIP EDIT
              ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
        END /* select */;
     CALL PLIRETC (RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E35X:   /* output handling routine prints sorted records*/
       PROC (INREC);
          DCL INREC CHAR(80);
          PUT SKIP EDIT (INREC) (A);
          CALL PLIRETC(4); /*request next record from sort*/
       END E35X;
    END EX108;
/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT   DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 53. PLISRTC—sorting from input data set to output handling routine

Calling PLISRTD example

```
 //OPT14#10  JOB ...
 //STEP1 EXEC IBMZCBG
 //PLI.SYSIN DD *
  EX109:  PROC OPTIONS(MAIN);
      DCL RETURN_CODE FIXED BIN(31,0);
      CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X,
                   E35X);

      SELECT(RETURN_CODE);
        WHEN(0)  PUT SKIP EDIT
            ('SORT COMPLETE RETURN_CODE 0') (A);
        WHEN(20) PUT SKIP EDIT
            ('SORT MESSAGE DATASET MISSING ') (A);
        OTHER    PUT SKIP EDIT
            ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
        END /* select */;

      CALL PLIRETC(RETURN_CODE);
       /*set PL/I return code to reflect success of sort*/

  E15X:   /* Input handling routine prints input before sorting*/
        PROC RETURNS(CHAR(80));
             DCL INFIELD CHAR(80);

             ON ENDFILE(SYSIN) BEGIN;
                PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT.  ',
                    'SORTED OUTPUT SHOULD FOLLOW')(A);
                CALL PLIRETC(8);  /* Signal end of input to sort*/
                GOTO ENDE15;
             END;

             GET FILE (SYSIN) EDIT (INFIELD) (A(80));
             PUT SKIP EDIT (INFIELD)(A);
             CALL PLIRETC(12);  /*Input to sort continues*/
            RETURN(INFIELD);
   ENDE15:
             END E15X;

   E35X:   /* Output handling routine prints the sorted records*/
        PROC (INREC);

             DCL INREC CHAR(80);
             PUT SKIP EDIT (INREC) (A);
      NEXT:  CALL PLIRETC(4); /* Request next record from sort*/
             END E35X;
  END EX109;
 /*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
```

Figure 54. PLISRTD—sorting from input handling routine to output handling routine

Sorting variable-length records example

```
//OPT14#11 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
  /* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH
     RECORDS */

  EX1306:  PROC OPTIONS(MAIN);
          DCL RETURN_CODE FIXED BIN(31,0);
          CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
                       ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                       /*NOTE THAT LENGTH IS MAX AND INCLUDES
                         4 BYTE LENGTH PREFIX*/
                       1048576
                       RETURN_CODE,
                       PUTIN,
                       PUTOUT);

               SELECT(RETURN_CODE);
                 WHEN(0)  PUT SKIP EDIT (
                        'SORT COMPLETE RETURN_CODE 0') (A);
                 WHEN(16) PUT SKIP EDIT (
                        'SORT FAILED, RETURN_CODE 16') (A);
                 WHEN(20) PUT SKIP EDIT (
                        'SORT MESSAGE DATASET MISSING ') (A);
                 OTHER    PUT SKIP EDIT (
                      'INVALID RETURN_CODE = ', RETURN_CODE)
                          (A,F(2));
               END /* SELECT */;

     CALL PLIRETC(RETURN_CODE);
      /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
      PUTIN: PROC RETURNS (CHAR(80) VARYING);
        /*OUTPUT HANDLING ROUTINE*/
        /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE
          WHEN USING VARYING LENGTH RECORDS*/
          DCL STRING CHAR(80) VAR;

          ON ENDFILE(SYSIN) BEGIN;
               PUT SKIP EDIT ('END OF INPUT')(A);
               CALL PLIRETC(8);
               GOTO ENDPUT;
               END;

          GET EDIT(STRING)(A(80));
          I=INDEX(STRING||'  ','  ')-1;/*RESET LENGTH OF THE*/
          STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO */
                                       /* LENGTH OF TEXT IN */
                                       /* EACH INPUT RECORD.*/
```

Figure 55 (Part 1 of 2). Sorting varying-length records using input and output handling routines

```
          PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
          CALL PLIRETC(12);
          RETURN(STRING);
 ENDPUT:  END;
 PUTOUT:PROC(STRING);
         /*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
          DCL STRING CHAR (*);
           /*NOTE THAT FOR VARYING RECORDS THE STRING
             PARAMETER FOR THE OUTPUT-HANDLING ROUTINE
             SHOULD BE DECLARED ADJUSTABLE BUT CANNOT BE
             DECLARED VARYING*/
          PUT SKIP EDIT(STRING)(A);  /*PRINT THE SORTED DATA*/
          CALL PLIRETC(4);
          END;  /*ENDS PUTOUT*/
         END;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//*
```

Figure 55 (Part 2 of 2). Sorting varying-length records using input and output handling routines

Part 6. Specialized programming tasks

Chapter 13. PL/I - Language Environment descriptors

This chapter describes PL/I parameter passing conventions between PL/I routines at run time. For additional information about Language Environment run-time environment considerations, other than descriptors, see *OS/390 Language Environment Programming Guide*. This includes run-time environment conventions and assembler macros supporting these conventions.

Passing an argument

When a string, an array, or a structure is passed as an argument, the compiler passes a descriptor for that argument unless the called routine is declared with OPTIONS(NODESCRIPTOR). There are two methods for passing such descriptors:

- By descriptor list
- By descriptor locator

The following key features should be noted about each of these two methods:

- **When arguments are passed with a descriptor list**
  - The number of arguments passed is one greater than the number of arguments specified if any of the arguments needs a descriptor.
  - An argument passed with a descriptor can be received as a pointer passed by value (BYVALUE).
- **When arguments are passed by descriptor locator**
  - The number of arguments passed always matches the number of arguments specified.
  - An argument passed with a descriptor can be received as a pointer passed by address (BYADDR).

Argument passing by descriptor list

When arguments and their descriptors are passed with a descriptor list, an extra argument is passed whenever at least one argument needs a descriptor. This extra argument is a pointer to a list of pointers. The number of entries in this list equals the number of arguments passed. For arguments that don't require a descriptor, the corresponding pointer in the descriptor list is set to SYSNULL. For arguments that do require a descriptor, the corresponding pointer in the descriptor list is set to the address of that argument's descriptor.

So, for example, suppose the routine `sample` is declared as

```
declare sample entry( fixed bin(31), varying char(*) )
               options( byaddr descriptor );
```

Then, if `sample` is called as in the following statement:

```
call sample( 1, 'test' );
```

The following three arguments are passed to the routine:

- Address of a fixed bin(31) temporary with the value 1
- Address of a varying char(4) temporary with the value `test`

- Address of a descriptor list consisting of the following:
  - SYSNULL()
  - Address of the descriptor for a varying char(4) string

Argument passing by descriptor-locator

When arguments and their descriptors are passed by descriptor-locator, whenever an argument requires a descriptor, the address of a locator/descriptor for it is passed instead.

The locator/descriptor is a pair of pointers. The first pointer is the address of the data; the second pointer is the address of the descriptor.

So, for example, suppose the routine `sample` is declared again as

```
declare sample entry( fixed bin(31), varying char(*) )
               options( byaddr descriptor );
```

Then, if `sample` is called as in the following statement

```
call sample( 1, 'test' );
```

The following two arguments are passed to the routine:

- Address of a fixed bin(31) temporary with the value 1
- Address of a descriptor-locator consisting of the following:
  - Address of a varying char(4) temporary with the value `test`
  - Address of the descriptor for a varying char(4) string

Descriptor header

Every descriptor starts with a 4-byte field. The first byte specifies the descriptor type (scalar, array, structure or union). The remaining three bytes are zero unless they are set by the particular descriptor type.

The declare for a descriptor header is:

```
declare
  1 dsc_Header based( sysnull() ),
    2 dsc_Type         fixed bin(8) unsigned,
    2 dsc_Datatype     fixed bin(8) unsigned,
    2 *                fixed bin(8) unsigned,
    2 *                fixed bin(8) unsigned;
```

The possible values for the dsc_Type field are:

```
declare
  dsc_Type_Unset             fixed bin(8) value(0),
  dsc_Type_Element           fixed bin(8) value(2),
  dsc_Type_Array             fixed bin(8) value(3),
  dsc_Type_Structure         fixed bin(8) value(4),
  dsc_Type_Union             fixed bin(8) value(4);
```

String descriptors

In a string descriptor, the second byte of the header indicates the string type (bit, character or graphic as well as nonvarying, varying or varyingz).

In a string descriptor for a nonvarying bit string, the third byte of the header gives the bit offset.

In a string descriptor for a varying string, the fourth byte has a bit indicating if the string length is held in nonnative format.

In a string descriptor for a character string, the fourth byte also has a bit indicating if the string data is in EBCDIC.

The declare for a string descriptor is:

```
declare
  1 dsc_String  based( sysnull() ),
    2 dsc_String_Header,
      3 *                fixed bin(8) unsigned,
      3 dsc_String_Type   fixed bin(8) unsigned,
      3 dsc_String_BitOfs fixed bin(8) unsigned,
      3 *,
        4 dsc_String_Has_Nonnative_Len    bit(1),
        4 dsc_String_Is_Ebcdic            bit(1),
        4 dsc_String_Has_Nonnative_Data   bit(1),
        4 *                               bit(5),
    2 dsc_String_Length   fixed bin(31); /* max length of string */
```

The possible values for the dsc_String_Type field are:

```
declare
  dsc_String_Type_Unset               fixed bin(8) value(0),
  dsc_String_Type_Char_Nonvarying     fixed bin(8) value(2),
  dsc_String_Type_Char_Varyingz       fixed bin(8) value(3),
  dsc_String_Type_Char_Varying2       fixed bin(8) value(4),
  dsc_String_Type_Bit_Nonvarying      fixed bin(8) value(6),
  dsc_String_Type_Bit_Varying2        fixed bin(8) value(7),
  dsc_String_Type_Graphic_Nonvarying fixed bin(8) value(9),
  dsc_String_Type_Graphic_Varyingz    fixed bin(8) value(10),
  dsc_String_Type_Graphic_Varying2    fixed bin(8) value(11),
  dsc_String_Type_Widechar_Nonvarying fixed bin(8) value(13),
  dsc_String_Type_Widechar_Varyingz   fixed bin(8) value(14),
  dsc_String_Type_Widechar_Varying2   fixed bin(8) value(15);
```

Array descriptors

The declare for an array descriptor is:

```
declare
  1 dsc_Array  based( sysnull() ),
    2 dsc_Array_Header   like dsc_Header,
    2 dsc_Array_EltLen   fixed bin(31), /* Length of array element */
    2 dsc_Array_Rank     fixed bin(31), /* Count of dimensions     */
    2 dsc_Array_RVO      fixed bin(31), /* Relative virtual origin */
    2 dsc_Array_Data( 1: 1 refer(dsc_Array_Rank) ),
      3 dsc_Array_LBound fixed bin(31), /*   LBound               */
      3 dsc_Array_Extent fixed bin(31), /*   HBound - LBound + 1   */
      3 dsc_Array_Stride fixed bin(31); /*   Multiplier           */
```

Chapter 14. Using PLIDUMP

This section provides information about dump options and the syntax used to call PLIDUMP, and describes PL/I-specific information included in the dump that can help you debug your routine.

Note: PLIDUMP conforms to National Language Support standards.

Figure 56 shows an example of a PL/I routine calling PLIDUMP to produce an Language Environment for OS/390 & VM dump. In this example, the main routine PLIDMP calls PLIDMPA, which then calls PLIDMPB. The call to PLIDUMP is made in routine PLIDMPB.

```
%PROCESS MAP GOSTMT SOURCE STG LIST OFFSET LC(101);
 PLIDMP: PROC  OPTIONS(MAIN) ;

   Declare   (H,I) Fixed bin(31) Auto;
   Declare   Names Char(17) Static init('Bob Teri Bo Jason');
   H = 5;  I = 9;
   Put skip list('PLIDMP Starting');
   Call PLIDMPA;

     PLIDMPA:  PROC;
       Declare  (a,b)  Fixed bin(31)   Auto;
       a = 1;  b = 3;
       Put skip list('PLIDMPA Starting');
       Call PLIDMPB;

         PLIDMPB:  PROC;
           Declare  1 Name  auto,
                      2  First   Char(12) Varying,
                      2  Last    Char(12) Varying;
           First = 'Teri';
           Last  = 'Gillispy';
           Put skip list('PLIDMPB Starting');
           Call PLIDUMP('TBFC','PLIDUMP called from procedure PLIDMPB');
           Put Data;
         End PLIDMPB;

     End PLIDMPA;

 End PLIDMP;
```

Figure 56. Example PL/I routine calling PLIDUMP

The syntax and options for PLIDUMP are shown below.

```
►►──PLIDUMP──(──character-string-expression 1──,──character-string-expression 2──►
►──)──────────────────────────────────────────────────────────────────────────────►◄
```

character-string-expression 1
is a dump options character string consisting of one or more of the following:

A	Requests information relevant to all tasks in a multitasking program.
B	BLOCKS (PL/I hexadecimal dump).
C	Continue. The routine continues after the dump.
E	Exit from current task of a multitasking program. Program continues to run after requested dump is completed.

F	FILES.
H	STORAGE. **Note:** A ddname of CEESNAP must be specified with the H option to produce a SNAP dump of a PL/I routine.
K	BLOCKS (when running under CICS). The Transaction Work Area is included.
NB	NOBLOCKS.
NF	NOFILES.
NH	NOSTORAGE.
NK	NOBLOCKS (when running under CICS).
NT	NOTRACEBACK.
O	Only information relevant to the current task in a multitasking program.
S	Stop. The enclave is terminated with a dump.
T	TRACEBACK.

T, F, and C are the default options.

character-string-expression 2
is a user-identified character string up to 80 characters long that is printed as the dump header.

PLIDUMP usage notes

If you use PLIDUMP, the following considerations apply:

- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- A DD statement with the ddname PLIDUMP, PL1DUMP, or CEEDUMP can be used to define the data set for the dump.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 133 to prevent dump records from wrapping.
- When you specify the H option in a call to PLIDUMP, the PL/I library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.
- Support for SNAP dumps using PLIDUMP is only provided under OS/390. SNAP dumps are not produced in a CICS environment.
  - If the SNAP is not successful, the CEE3DMP DUMP file displays the message:

    ```
    Snap was unsuccessful
    ```

  - If the SNAP is successful, CEE3DMP displays the message:

    ```
    Snap was successful; snap ID = nnn
    ```

where *nnn* corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

If you want to ensure portability across system platforms, use PLIDUMP to generate a dump of your PL/I routine.

Chapter 15. Interrupts and attention processing

To enable a PL/I program to recognize attention interrupts, two operations must be possible:

- You must be able to create an interrupt. This is done in different ways depending upon both the terminal you use and the operating system.
- Your program must be prepared to respond to the interrupt. You can write an ON ATTENTION statement in your program so that the program receives control when the ATTENTION condition is raised.

  Note: If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following:

  – The INTERRUPT option (supported only in TSO)
  – A TEST option other than NOTEST or TEST(NONE,NOSYM).

  Compiling this way causes INTERRUPT(ON) to be in effect, unless you explicitly specify INTERRUPT(OFF) in PLIXOPT.

You can find the procedure used to create an interrupt in the IBM instruction manual for the operating system and terminal that you are using.

There is a difference between the interrupt (the operating system recognized your request) and the raising of the ATTENTION condition.

An *interrupt* is your request that the operating system notify the running program. If a PL/I program was compiled with the INTERRUPT compile-time option, instructions are included that test an internal interrupt switch at discrete points in the program. The internal interrupt switch can be set if any program in the load module was compiled with the INTERRUPT compile-time option.

The internal switch is set when the operating system recognizes that an interrupt request was made. The execution of the special testing instructions (polling) raises the ATTENTION condition. If a debugging tool hook (or a CALL PLITEST) is encountered before the polling occurs, the debugging tool can be given control before the ATTENTION condition processing starts.

Polling ensures that the ATTENTION condition is raised between PL/I statements, rather than within the statements.

Figure 57 on page 245 shows a skeleton program, an ATTENTION ON-unit, and several situations where polling instructions will be generated. In the program polling will occur at:

- LABEL1
- Each iteration of the DO
- The ELSE PUT SKIP ... statement
- Block END statements

```
%PROCESS INTERRUPT;
     .
     .
     .
 ON ATTENTION
   BEGIN;
     DCL X FIXED BINARY(15);
     PUT SKIP LIST ('Enter 1 to terminate, 0 to continue.');
     GET SKIP LIST (X);
     IF X = 1 THEN
       STOP;
     ELSE
       PUT SKIP LIST ('Attention was ignored');
   END;
     .
     .
     .
  LABEL1:
   IF EMPNO ...
     .
     .
     .
   DO I = 1 TO 10;
     .
     .
     .
   END;
     .
     .
     .
```

Figure 57. Using an ATTENTION ON-unit

Using ATTENTION ON-units

You can use processing within the ATTENTION ON-unit to terminate potentially endless looping in a program.

Control is given to an ATTENTION ON-unit when polling instructions recognize that an interrupt has occurred. Normal return from the ON-unit is to the statement following the polling code.

Interaction with a debugging tool

If the program has the TEST(ALL) or TEST(ERROR) run-time option in effect, an interrupt causes the debugging tool to receive control the next time a hook is encountered. This might be before the program's polling code recognizes that the interrupt occurred.

Later, when the ATTENTION condition is raised, the debugging tool receives control again for condition processing.

Chapter 16. Using the Checkpoint/Restart facility

This chapter describes the PL/I Checkpoint/Restart feature which provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment.

At points specified in the program, information about the current status of the program is written as a record on a data set. If the program terminates due to a system failure, you can use this information to restart the program close to the point where the failure occurred, avoiding the need to rerun the program completely.

This restart can be either automatic or deferred. An automatic restart is one that takes place immediately (provided the operator authorizes it when requested by a system message). A deferred restart is one that is performed later as a new job.

You can request an automatic restart from within your program without a system failure having occurred.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system. This facility is described in the books listed in "Bibliography" on page 259.

To use checkpoint/restart you must do the following:

- Request, at suitable points in your program, that a checkpoint record is written. This is done with the built-in subroutine PLICKPT.
- Provide a data set on which the checkpoint record can be written.
- Also, to ensure the desired restart activity, you might need to specify the RD parameter in the EXEC or JOB statement (see *OS/390 JCL Reference*).

Note: You should be aware of the restrictions affecting data sets used by your program. These are detailed in the "Bibliography" on page 259.

Requesting a checkpoint record

Each time you want a checkpoint record to be written, you must invoke, from your PL/I program, the built-in subroutine PLICKPT.

```
►►──CALL──PLICKPT──┬─────────────────────────────────────────────────────┬──►◄
                   └─(──ddname──┬──────────────────────────────────────┬─)─┘
                                └─,──check-id──┬──────────────────────┬┘
                                               └─,──org──┬──────────┬─┘
                                                         └─,──code──┘
```

The four arguments are all optional. If you do not use an argument, you need not specify it unless you specify another argument that follows it in the given order. In this case, you must specify the unused argument as a null string (''). The following paragraphs describe the arguments.

ddname
: is a character string constant or variable specifying the name of the DD statement defining the data set that is to be used for checkpoint records. If you omit this argument, the system will use the default ddname SYSCHK.

check-id
is a character string constant or variable specifying the name that you want to assign to the checkpoint record so that you can identify it later. If you omit this argument, the system will supply a unique identification and print it at the operator's console.

org
is a character string constant or variable with the attributes CHARACTER(2) whose value indicates, in operating system terms, the organization of the checkpoint data set. PS indicates sequential (that is, CONSECUTIVE) organization; PO represents partitioned organization. If you omit this argument, PS is assumed.

code
is a variable with the attributes FIXED BINARY (31), which can receive a return code from PLICKPT. The return code has the following values:

0 A checkpoint has been successfully taken.

4 A restart has been successfully made.

8 A checkpoint has not been taken. The PLICKPT statement should be checked.

12 A checkpoint has not been taken. Check for a missing DD statement, a hardware error, or insufficient space in the data set. A checkpoint will fail if taken while a DISPLAY statement with the REPLY option is still incomplete.

16 A checkpoint has been taken, but ENQ macro calls are outstanding and will not be restored on restart. This situation will not normally arise for a PL/I program.

Defining the checkpoint data set

You must include a DD statement in the job control procedure to define the data set in which the checkpoint records are to be placed. This data set can have either CONSECUTIVE or partitioned organization. You can use any valid ddname. If you use the ddname SYSCHK, you do not need to specify the ddname when invoking PLICKPT.

You must specify a data set name only if you want to keep the data set for a deferred restart. The I/O device can be any direct-access device.

To obtain only the last checkpoint record, then specify status as NEW (or OLD if the data set already exists). This will cause each checkpoint record to overwrite the previous one.

To retain more than one checkpoint record, specify status as MOD. This will cause each checkpoint record to be added after the previous one.

If the checkpoint data set is a library, “check-id” is used as the member-name. Thus a checkpoint will delete any previously taken checkpoint with the same name.

For direct-access storage, you should allocate enough primary space to store as many checkpoint records as you will retain. You can specify an incremental space allocation, but it will not be used. A checkpoint record is approximately 5000 bytes longer than the area of main storage allocated to the step.

No DCB information is required, but you can include any of the following, where applicable:

```
OPTCD=W, OPTCD=C, RECFM=UT
```

These subparameters are described in the *OS/390 JCL User's Guide*.

Requesting a restart

A restart can be automatic or deferred. You can make automatic restarts after a system failure or from within the program itself. The system operator must authorize all automatic restarts when requested by the system.

Automatic restart after a system failure

If a system failure occurs after a checkpoint has been taken, the automatic restart will occur at the last checkpoint if you have specified RD=R (or omitted the RD parameter) in the EXEC or JOB statement.

If a system failure occurs before any checkpoint has been taken, an automatic restart, from the beginning of the job step, can still occur if you have specified RD=R in the EXEC or JOB statement.

After a system failure occurs, you can still force automatic restart from the beginning of the job step by specifying RD=RNC in the EXEC or JOB statement. By specifying RD=RNC, you are requesting an automatic step restart without checkpoint processing if another system failure occurs.

Automatic restart within a program

You can request a restart at any point in your program. The rules for the restart are the same as for a restart after a system failure. To request the restart, you must execute the statement:

```
CALL PLIREST;
```

To effect the restart, the compiler terminates the program abnormally, with a system completion code of 4092. Therefore, to use this facility, the system completion code 4092 must not have been deleted from the table of eligible codes at system generation.

Getting a deferred restart

To ensure that automatic restart activity is canceled, but that the checkpoints are still available for a deferred restart, specify RD=NR in the EXEC or JOB statement when the program is first executed.

```
►►──RESTART──=──(──stepname──┬─────────────┬──)──────────────────────────────►◄
                             └─,──check-id─┘
```

If you subsequently require a deferred restart, you must submit the program as a new job, with the RESTART parameter in the JOB statement. Use the RESTART parameter to specify the job step at which the restart is to be made and, if you want to restart at a checkpoint, the name of the checkpoint record.

For a restart from a checkpoint, you must also provide a DD statement that defines the data set containing the checkpoint record. The DD statement must be named SYSCHK. The DD statement must occur immediately before the EXEC statement for the job step.

Modifying checkpoint/restart activity

You can cancel automatic restart activity from any checkpoints taken in your program by executing the statement:

```
CALL PLICANC;
```

However, if you specified RD=R or RD=RNC in the JOB or EXEC statement, automatic restart can still take place from the beginning of the job step.

Also, any checkpoints already taken are still available for a deferred restart.

You can cancel any automatic restart and the taking of checkpoints, even if they were requested in your program, by specifying RD=NC in the JOB or EXEC statement.

Chapter 17. Using user exits

PL/I provides a number of user exits that allow you to customize the PL/I product to suit your needs. The PL/I products supply default exits and the associated source files.

If you want the exits to perform functions that are different from those supplied by the default exits, we recommend that you modify the supplied source files as appropriate.

At times, it is useful to be able to tailor the compiler to meet the needs of your organization. For example, you might want to suppress certain messages or alter the severity of others. You might want to perform a specific function with each compilation, such as logging statistical information about the compilation into a file. A compiler user exit handles this type of function.

With PL/I, you can write your own user exit or use the exit provided with the product, either `'as is'` or modified, depending on what you want to do with it. The 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 58, 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.

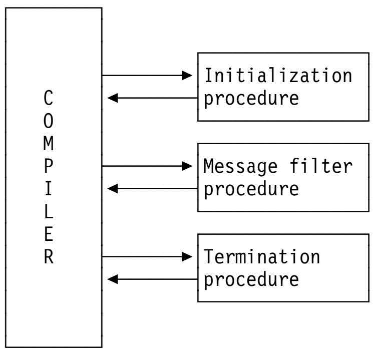


Figure 58. PL/I compiler user exit procedures

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 252.
- 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 254, “Writing the message filtering procedure” on page 254, and “Writing the termination procedure” on page 255.

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 14.

The EXIT compile-time option allows you to specify a user-option-string which specifies the DDname for the user exit input file. If you do not specify a string, SYSUEXIT is used as the DDname for the user exit input file.

The user-option-string is passed to the user exit functions in the global control block which is discussed in “Structure of global control blocks” on page 252. Please refer to the field “Uex_UIB_User_char_str” in the section “Structure of global control blocks” on page 252 for additional information.

The IBM-supplied compiler exit, IBMUEXIT

IBM supplies you with the sample compiler user exit, IBMUEXIT, which filters messages for you. It monitors messages and, based on the message number that you specify, suppresses the message or changes the severity of the message.

Customizing the compiler user exit

As was mentioned earlier, you can write your own compiler user exit or simply use the one shipped with the compiler. In either case, the name of the fetchable file for the compiler user exit must be IBMUEXIT.

This section describes how to:

- Modify the user exit input file for customized message filtering
- Create your own compiler user exit

Modifying SYSUEXIT

Rather than spending the time to write a completely new compiler user exit, you can simplify modify the user exit input file.

Edit the file to indicate which message numbers you want to suppress, and which message number severity levels you would like changed. A sample file is shown in Figure 59.

```
  Fac Id   Msg No   Severity   Suppress   Comment
+--------+--------+----------+----------+--------------------------------
  'IBM'     1042      -1         1      String spans multiple lines
  'IBM'     1044      -1         1      FIXED BIN 7 mapped to 1 byte
  'IBM'     1047       8         0      Order inhibits optimization
  'IBM'     1052      -1         1      Nodescriptor with * extent arg
  'IBM'     1059       0         0      Select without OTHERWISE
  'IBM'     1169       0         1      Precision of result determined
```

Figure 59. Example of an user exit input file

The first two lines are header lines and are ignored by IBMUEXIT. The remaining lines contain input separated by a variable number of blanks.

Each column of the file is relevant to the compiler user exit:

- The first column 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 (see the source in Figure 16.) as a model. As you write the exit, make sure it covers the areas of initialization, message filtering, and termination.

Structure of global control blocks

The global control block is passed to each of the three user exit procedures (initialization, filtering, and termination) whenever they are invoked. The following code and accompanying explanations describe the contents of each field in the global control block.

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

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

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

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

    2 Uex_UIB_return_code fixed bin(31),    /* set by exit procs   */
    2 Uex_UIB_reason_code fixed bin(31),    /* set by exit procs   */

    2 Uex_UIB_Exit_Routs,                   /* exit entries set at
                                               initialization      */
      3 ( Uex_UIB_Termination,
          Uex_UIB_Message_Filter,           /* call for each msg   */
          *, *, *, * )
        limited entry (
             *,                              /* to Uex_UIB          */
             *,                              /* to a request area   */
         );
```

Data Entry Fields

- **Uex_UIB_ Length**: Contains the length of the control block in bytes. The value is `storage (Uex_UIB)`.
- **Uex_UIB_Exit_token**: Used by the user exit procedure. For example, the initialization may set it to a data structure which is used by both the message filter, and the termination procedures.
- **Uex_UIB_User_char_str**: Points to an optional character string, if you specify it. For example, in `pli filename (EXIT ('string'))...fn` can be a character string up to thirty-one characters in length.
- **Uex_UIB_char_len**: Contains the length of the string pointed to by the `User_char_str`. The compiler sets this value.
- **Uex_UIB_Filename_str**: Contains the name of the source file that you are compiling, and includes the drive and subdirectories as well as the filename. The compiler sets this value.
- **Uex_UIB_Filename_len**: Contains the length of the name of the source file pointed to by the `Filename_str`. The compiler sets this value.
- **Uex_UIB_return_code**: Contains the return code from the user exit procedure. The user sets this value.
- **Uex__UIB_reason_code**: Contains the procedure reason code. The user sets this value.
- **Uex_UIB_Exit_Routs**: Contains the exit entries set up by the initialization procedure.
- **Uex_UIB_Termination**: Contains the entry that is to be called by the compiler at termination time. The user sets this value.

- **Uex_UIB_Message_Filter**: Contains the entry that is to be called by the compiler whenever a message needs to be generated. The user sets this value.

Writing the initialization procedure

Your initialization procedure should perform any initialization required by the exit, such as opening files and allocating storage. The initialization procedure-specific control block is coded as follows:

```
Dcl 1 Uex_ISA native based( null() ),
    2 Uex_ISA_Length_fixed bin(31);   /* storage(Uex_ISA) *  /
```

The global control block syntax for the initialization procedure is discussed in the section "Structure of global control blocks" on page 252.

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 252. 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

Appendix. 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:

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM VisualAge PL/I for OS/390.

Macros for customer use

IBM VisualAge PL/I for OS/390 provides no macros that allow a customer installation to write programs that use the services of IBM VisualAge PL/I for OS/390.

Attention: Do not use as programming interfaces any IBM VisualAge PL/I for OS/390 macros.

Trademarks

Bibliography

VisualAge PL/I for OS/390 publications

Fact Sheet, GC26-9470

Licensed Program Specifications, GC26-9471

Programming Guide, SC26-9473

Compiler and Run-Time Migration Guide, SC26-9474

Diagnosis Guide, SC26-9475

Language Reference, SC26-9476

Messages and Codes, SC26-9478

Other PL/I publications

PL/I for MVS & VM

Installation and Customization under MVS, SC26-3119

Programming Guide, SC26-3113

Language Reference, SC26-3114

Reference Summary, SX26-3821

Compiler and Run-Time Migration Guide, SC26-3118

Compile-Time Message and Codes, SC26-3229

Diagnosis Guide, SC26-3149

VisualAge PL/I Enterprise (OS/2® and Windows®)

Fact Sheet, GC26-9187

Programming Guide, GC26-9177

Language Reference, GC26-9178

Messages and Codes, GC26-9179

Building GUIs on OS/2, GC26-9180

PL/I Set for AIX®

Programming Guide, SC26-8456

Language Reference, SC26-8455

Messages and Codes, SC26-8457

Program Builder User's Guide, SC09-2201

LPEX User's Guide and Reference, SC09-2202

OS/390 Language Environment publications

Concepts Guide, GC28-1945

Programming Guide, SC28-1939

Programming Reference, SC28-1940

Customization, SC28-1941

Debugging Guide and Run-Time Messages, SC28-1942

Run-Time Migration Guide, SC28-1944

Writing Interlanguage Applications, SC28-1943

IBM Debug Tool publications

User's Guide and Reference, SC09-2137

Reference Summary, SX26-3840

Softcopy publications

Online publications are distributed on CD-ROMs and can be ordered from Mechanicsburg through your IBM representative. PL/I books are distributed on the following collection kits:

OS/390 Collection Kit, SK2T-6700
Messages & Codes Collection Kit, SK2T-2068

Other books you might need

CICS/ESA®

Application Programming Guide, SC33-1169

Application Programming Reference, SC33-1170

Sample Applications Guide, SC33-1173

DFSORT™

Application Programming Guide, SC33-4035

Installation and Customization, SC33-4034

IMS/ESA®

Application Programming: Database Manager, SC26-8015

Application Programming: Database Manager Summary, SC26-8037

Application Programming: Design Guide, SC26-8016

Application Programming: Transaction Manager, SC26-8017

Application Programming: Transaction Manager Summary, SC26-8038

Application Programming: EXEC DL/I Commands for CICS and IMS™, SC26-8018

Application Programming: EXEC DL/I Commands for CICS and IMS Summary, SC26-8036

OS/390 TSO/E

User's Guide, SC28-1968

Command Reference, SC28-1969

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

DDM file. A &system. file that is associated with a remote file that is accessed using DDM. The DDM file provides the information needed for a local (source) system to locate a remote (target) system and to access the file at the target system where the requested data is stored.

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

We'd Like to Hear from You

IBM VisualAge PL/I for OS/390
Programming Guide
Version 2 Release 2.1

Publication No. SC26-9473-01

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

IBM VisualAge PL/I for OS/390
Programming Guide
Version 2 Release 2.1

Publication No. SC26-9473-01

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
SC26-9473-01

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/H3
PO Box 49023
San Jose, CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

SC26-9473-01

Cut or Fold
Along Line

IBM®

Program Number: 5655-B22

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

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

Common VisualAge PL/I Library

SC26-9476	Language Reference

SC26-9473-01

Spine information:

IBM VisualAge PL/I for OS/390 Programming Guide *Version 2 Release 2.1*