IBM PL/I for VSE/ESA

Programming Guide

Release 1

SC26-8053-00

IBM PL/I for VSE/ESA

Programming Guide

Release 1

SC26-8053-00

Note!

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

First Edition (April 1995)

This edition applies to Version 1 Release 1 of IBM PL/I for VSE/ESA, 5686-069, 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 J58
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

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Programming interface information

This book is intended to help the customer write programs using IBM PL/I for VSE/ESA. This book documents General-use Programming Interface and Associated Guidance Information provided by IBM PL/I for VSE/ESA.

General-use programming interfaces allow the customer to write programs that obtain the services of IBM PL/I for VSE/ESA.

Macros for customer use

IBM PL/I for VSE/ESA provides no macros that allow a customer installation to write programs that use the services of IBM PL/I for VSE/ESA.

Warning: Do not use as programming interfaces any IBM PL/I for VSE/ESA macros.

Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

AD/Cycle
C/370
CICS
ECKD
IBM
Language Environment
OS/2
SAA
VSE/ESA

Part 1. Introduction

About this book

This book is for PL/I programmers and systems programmers. It helps you understand how to use IBM* PL/I for VSE/ESA* (PL/I VSE) to compile PL/I programs. It also describes the operating system features that you might need to optimize program performance or handle errors.

Using your documentation

The publications provided with PL/I VSE are designed to help you do PL/I programming under VSE. Each publication helps you perform a different task.

Where to look for more information

For information about the PL/I VSE library, see Table 1.

Table 1. How to use the publications you receive with PL/I VSE

To...	Use...
Evaluate the product	*Fact Sheet*
Understand warranty information	*Licensed Program Specifications*
Install the compiler	*Installation and Customization Guide*
Understand product changes and adapt programs to PL/I VSE	*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* *Reference Summary*
Diagnose compiler problems and report them to IBM	*Diagnosis Guide*
Get details on compile-time messages[1]	*Compile-Time Messages and Codes*

Note:

1. For details on run-time messages, see the LE/VSE library.

You might also require information about IBM* Language Environment* for VSE/ESA* (LE/VSE). For information about the LE/VSE library, see Table 2.

Table 2 (Page 1 of 2). How to use the publications you receive with LE/VSE

To...	Use...
Evaluate Language Environment	*Fact Sheet* *Concepts Guide*
Install LE/VSE	*Installation and Customization Guide*
Understand the LE/VSE program models and concepts	*Concepts Guide* *Programming Guide*
Prepare your LE/VSE-conforming applications and find syntax for run-time options and callable services	*Programming Guide* *Reference Summary*
Debug your LE/VSE-conforming application and get details on run-time messages	*Debugging Guide and Run-Time Messages*

Table 2 (Page 2 of 2). How to use the publications you receive with LE/VSE

To...	Use...
Diagnose problems that occur in your LE/VSE-conforming application	*Diagnosis Guide*
Understand warranty information	*Licensed Program Specifications*

For the complete titles and order numbers of these and other related publications, see the "Bibliography" on page 343.

What is new in PL/I VSE

This is a major new release of PL/I, containing many new features and facilities. It brings to VSE many of the functions of the MVS & VM version of PL/I (IBM SAA* AD/Cycle* PL/I MVS & VM), while retaining close source compatibility with the DOS PL/I Optimizing Compiler (DOS PL/I).

PL/I VSE enables you to integrate your PL/I applications into IBM Language Environment for VSE/ESA (LE/VSE). In addition to PL/I's already impressive features, you gain access to LE/VSE's rich set of library routines and enhanced interlanguage communication (ILC) with IBM COBOL for VSE/ESA (COBOL/VSE).

IBM Language Environment for VSE/ESA support

PL/I VSE provides the following functions in the LE/VSE area:

Interlanguage communication (ILC) support:

- Object code produced by PL/I VSE Release 1 can be linked with object code produced by other LE/VSE-conforming compilers (currently only COBOL/VSE).
- PL/I VSE programs can fetch COBOL/VSE phases.
- COBOL/VSE programs can fetch PL/I VSE phases.

Note: PL/I VSE does not support ILC with:

- FORTRAN
- RPG
- DOS/VS COBOL
- C/370*

Limited ILC support is provided for VS COBOL II at Release 3.2 or later.

Common support for multiple operating environments:

- Some of the restrictions on PL/I coding in the CICS* environment have been lifted.
- Procedure OPTIONS option FETCHABLE can be used to specify the procedure that gets control within a fetched phase.
- CEETDLI is supported in addition to PLITDLI and EXEC DLI.
- LE/VSE services provide storage management and condition handling support, as well as PLIDUMP and MSGFILE support for PL/I messages and other output.
- By default, only user-generated output is written to SYSLST. All run-time generated messages are written to MSGFILE.

- ERROR conditions now get control of all system abends. The PL/I message is issued only if there is no ERROR on-unit or if the ERROR on-unit does not recover from the condition via a GOTO.
- Selected items from PL/I Package/2 (the PL/I product for OS/2*) are implemented to allow better coexistence.
  - Limited support of OPTIONS(BYVALUE and BYADDR)
  - Limited support of EXTERNAL(environment-name) allowing alternate external names
  - Limited support of OPTIONAL arguments/parameters
  - Support for %PROCESS statement
  - NOT and OR compiler options

Product packaging:

- All PL/I VSE resident library routines are now packaged with LE/VSE, and are loaded at run time rather than link-edited with the application program. Changes to the resident library no longer require PL/I programs to be re-linked.
- At link-edit time, you have the option of getting math results that are compatible with LE/VSE or with DOS PL/I.
- Installation enhancements are provided to ease product installation and migration.

For migration considerations, see the *PL/I VSE Migration Guide.*

Usability enhancements

These enhancements expand the PL/I language statements and options, PL/I data types, and compiler options, to make the language easier to use.

Enhanced double-byte character set (DBCS) support: This support introduces many enhancements that facilitate processing of GRAPHIC and mixed-character data and allows the source of the PL/I program to be in DBCS and/or the single-byte character set (SBCS), rather than only in SBCS.

Hexadecimal data constants: Constants for bit and character data can now be defined in hexadecimal notation, such that each *character* (0-9 and A-F) represents 4 bits.

Interface improvements for all (sub)systems: A new compiler option, SYSTEM, lets the programmer specify the target operating environment (of the generated object code), and the format of the parameters for the MAIN procedure.

Specification of compile-time options: You can specify compile-time options on the *PROCESS statement, a new %PROCESS statement, and in the PARM option of the EXEC IEL1AA JCL statement.

Linking after errors: The COMPILE compile-time option has been enhanced to allow linking to proceed after a severe error.

Run-time options: You can specify program run-time options in the PARM option of the EXEC JCL statement. PL/I VSE and LE/VSE will use these to control the execution of PL/I programs.

Passing parameters to the MAIN procedure: VSE JCL can also be used to pass a parameter to the MAIN PL/I procedure. A slash (/) separates the run-time options from the program parameter.

OPEN statement enhancements:

- There are new parameters on the PL/I OPEN statement that allow additional file attributes to be specified at file open time. These attributes are added to those in the file declaration.
- A vendor exit on the PL/I OPEN statement can be used to change the system logical unit number of the PL/I spill file.
- Data set name sharing for VSAM files, using the DSN option of the ENVIRONMENT attribute.

Date and time enhancements: A new built-in function, DATETIME, returns consistent date and time, including the four-digit year.

PL/I statement numbering options: A new compiler option, NUMBER, specifies that PL/I statement numbers will be derived from the sequence numbers in the program source deck, instead of being allocated sequentially.

Dynamic loading of external procedures: PL/I now supports the FETCH and RELEASE statements, to load external procedures into main storage at run time instead of having them link-edited with the MAIN procedure. (If these external procedures are PL/I, they must be compiled with PL/I VSE.)

New I/O facilities: PL/I VSE provides the following new I/O facilities:

- Support for REGIONAL(2) files
- Support for V and VS formats on REGIONAL(3) files
- Support for DELETE statement on REGIONAL files
- Support for multitrack search on REGIONAL files, using the LIMCT option of the ENVIRONMENT attribute
- Support for VSAM variable-length relative-record data sets (VRDS)
- Support for V format on consecutive unbuffered files
- Support for VS and VBS formats on consecutive buffered files

System programmer functions: A number of significant new features enhance PL/I as a system programming language:

- Support of additional program execution environments.

  PL/I can now be used for some system exit routines, such as the LE/VSE initialization exit.

- Additional support for pointers.

  PL/I built-in functions are now available to perform extended operations on pointers, including pointer arithmetic.

- Additional support for entry variables.

  A new built-in function and pseudovariable, ENTRYADDR, allows programmers to manipulate entry point addresses of procedures.

Extended addressing enhancements

These enhancements exploit the large amounts of storage available in the VSE/ESA environment, making programming easier.

Addressing mode: PL/I VSE programs can be link-edited with AMODE(31) and RMODE(ANY).

Location of variables: PL/I variables can now be located above the 16-megabyte line.

Fullword array subscripts: Array bounds can now be in the range -2^{31} (-2,147,483,648) through $+2^{31}-1$ (+2,147,483,647). The associated built-in functions (such as LBOUND and HBOUND) now return FIXED BINARY(31) values.

AREA and aggregate sizes: An AREA can now have a maximum size of 2,147,483,647 ($2^{31}-1$) bytes.

An aggregate can now have a maximum size of 2,147,483,647 ($2^{31}-1$) bytes. For unaligned BIT arrays and aggregates that contain any unaligned BIT data (arrays or non-arrays), the maximum size is 268,435,455 ($2^{28}-1$) bytes.

These numbers include any control information bytes that might be needed.

Notation conventions used in this book

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

Conventions used

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

- Items shown in UPPERCASE letters indicate key elements that must be typed exactly as shown.
- Items shown in lowercase letters indicate user-supplied variables for which you must substitute appropriate names or values. The variables begin with a letter and can include hyphens, numbers, or the underscore character (_).
- The term *digit* indicates that a digit (0 through 9) should be substituted.
- The term *do-group* indicates that a do-group should be substituted.
- Underlined items indicate default options.
- `Examples` are shown in monospace 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 xviii, are part of the programming syntax itself.

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

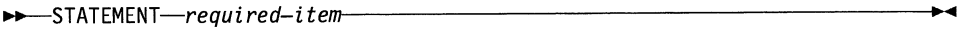
How to read the syntax notation

Throughout this book, syntax is described using the following structure:

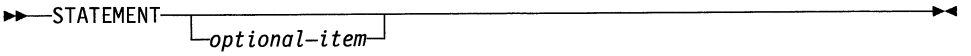
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following table shows the meaning of symbols at the beginning and end of syntax diagram lines.

Symbol	Indicates
▶▶—	the syntax diagram starts here
—▶	the syntax diagram is continued on the next line
▶—	the syntax diagram is continued from the previous line
—▶◀	the syntax diagram ends here

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



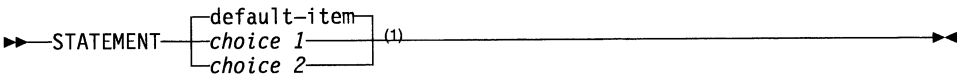
- Optional items appear below the main path.



- When you can choose from two or more items, the items appear vertically, in a stack. If you **must** choose one of the items, one item of the stack appears on the main path. The default, if any, appears above the main path and is chosen by the compiler if you do not specify another choice.

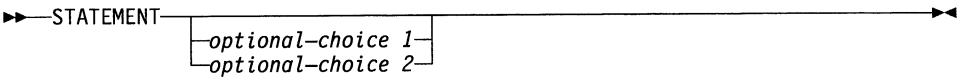
  Note: In some cases, the default is affected by the:

  - System in which the program is being run
  - Environmental parameters specified

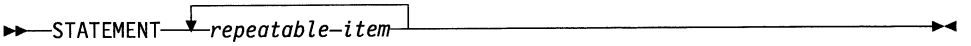


  Note:
  [1] Because *choice 1* appears on the horizontal bar, one of the items must be included in the statement. If you don't specify either *choice 1* or *choice 2*, the compiler implements the default for you.

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

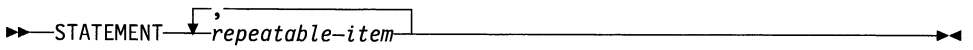


- An arrow returning to the left above the main line is a *repeat arrow*, and it indicates an item that can be repeated.



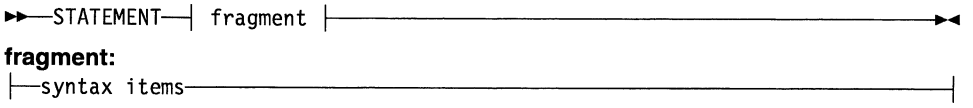
  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- If there is a comma as part of the repeat arrow, you must use a comma to separate items in a series.



If the comma appears below the repeat arrow line instead of on the line as shown in the previous example, the comma is optional as a separator of items in a series.

- A syntax fragment is delimited in the main syntax diagram by a set of vertical lines. The corresponding meaning of the fragment begins with the name of the fragment followed by the syntax, which starts and ends with a vertical line.



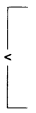
- Keywords appear in uppercase (for example, STATEMENT). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *item*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other symbols are shown, you must enter them as part of the syntax.

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.

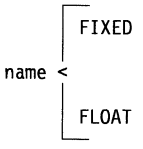
- **Braces**, { }, indicate a choice of entry. Unless an item is underlined, indicating a default, or the items are enclosed in brackets, you must choose at least one of the entries.

  The symbol:



  represents a large brace. When items are stacked within a large brace, choose only one of the items.

  For example:



  indicates that the variable *name* must be followed by FIXED or FLOAT. FIXED and FLOAT must be typed exactly as shown.

- 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 VSE 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 this section.

```
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.

Your installation can change the IBM-supplied defaults when this product is installed. Therefore, the defaults listed in this chapter might not be the same as those chosen by your installation. You can override most defaults when you compile your PL/I program.

Compile-time options

There are three types of compile-time options:

1. Simple pairs of keywords: a positive form requesting a facility, and an alternative negative form inhibiting 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. Table 4 on page 6 lists the options by function so that you can more easily see the types of facilities that are available.

The paragraphs following Table 3 and Table 4 describe the options in alphabetical order. For those options that specify 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 32.

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

Compile-time option	Abbreviated name	IBM default
AGGREGATE|NOAGGREGATE	AG|NAG	NAG
ATTRIBUTES[(FULL|SHORT)]| NOATTRIBUTES	A[(FS)]|NA	NA [(FULL)][1]
CMPAT(V1|V2)[2]	CMP(V1|V2)	–
COMPILE|NOCOMPILE[(W|E|S)]	CINC[(W|E|S)]	NC(S)
CONTROL('password')	–	–
DECK|NODECK[3]	DIND	–
ESD|NOESD	–	NOESD
FLAG[(I|W|E|S)]	F[(I|W|E|S)]	F(I)
GONUMBER|NOGONUMBER	GN|NGN	NGN
GOSTMT|NOGOSTMT	GS|NGS	NGS
GRAPHIC|NOGRAPHIC	GR|NGR	NGR
IMPRECISE|NOIMPRECISE[2]	IMP|NIMP	–
INCLUDE|NOINCLUDE	INC|NINC	NINC
INSOURCE|NOINSOURCE	IS|NIS	IS
INTERRUPT|NOINTERRUPT[2]	INT|NINT	–
LANGLVL({OS,SPROG|NOSPROG})	–	LANGLVL(OS,NOSPROG)
LINECOUNT(n)	LC(n)	LC(55)
LIST[(m[,n])]|NOLIST	–	NOLIST

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

Compile-time option	Abbreviated name	IBM default
LMESSAGE|SMESSAGE	LMSG|SMSG	LMSG
MACRO|NOMACRO	MI|NM	NM
MAP|NOMAP	–	NOMAP
MARGINI('c')|NOMARGINI	MI('c')|NMI	NMI
MARGINS(m,n[,c])	MAR(m,n[,c])	MAR(2,72)
MDECK|NOMDECK	MD|NMD	NMD
NAME('name[,origin[,NOAUTO]]')	N('name[,origin[,NOAUTO]]')	–
NOT	–	NOT('¬')
NEST|NONEST	–	NONEST
NUMBER|NONUMBER	NUM|NNUM	NNUM
OBJECT|NOOBJECT[4]	OBJ|NOBJ	–
OFFSET|NOOFFSET	OF|NOF	NOF
OPTIMIZE(TIME|0|2)|NOOPTIMIZE	OPT(TIME|0|2)|NOPT	NOPT
OPTIONS|NOOPTIONS	OP|NOP	OP
OR	–	OR('|')
SEQUENCE(m,n)|NOSEQUENCE	SEQ(m,n)|NSEQ	SEQ(73,80)
SIZE([-]yyyyyyyy|[-]yyyyyK|MAX)	SZ([-]yyyyyyyy|[-]yyyyyK|MAX)	SZ(MAX)
SOURCE|NOSOURCE	S|NS	S
STMT|NOSTMT	–	STMT
STORAGE|NOSTORAGE	STG|NSTG	NSTG
SYNTAX|NOSYNTAX[(W|E|S)]	SYN|NSYN[(W|E|S)]	NSYN(S)
SYSTEM(VSE|CICS|DLI|DL1)	–	SYSTEM(VSE)
TERMINAL[(opt-list)]|NOTERMINAL[2]	TERM[(opt-list)]|NTERM	–
TEST[([ALL|BLOCK|NONE|PATH| STMT][,SYM|,NOSYM])]|NOTEST	–	NOTEST [(NONE,SYM)][5]
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. The CMPAT, INTERRUPT, and TERMINAL options have no effect. They are retained for compatibility with the MVS and VM implementations of the compiler.
3. The DECK\|NODECK option can only be specified via the JCL OPTION statement (OPTION DECK). It will be ignored if specified as a PL/I compiler option.
4. The OBJECT\|NOOBJECT option can only be specified via the JCL OPTION statement (OPTION LINK or OPTION CATAL). It will be ignored if specified as a PL/I compiler option.
5. (NONE,SYM) is the default suboption if the suboption is omitted with TEST.

Table 4 (Page 1 of 2). Compile-time options arranged by function

Listing options		
Control listings produced	**AGGREGATE**	lists aggregates and their size.
	ATTRIBUTES	lists attributes of identifiers.
	ESD	lists external symbol dictionary.
	FLAG	suppresses diagnostic messages below a certain severity.
	INSOURCE	lists preprocessor input.
	LIST	lists object code produced by compiler.
	LMESSAGE/SMESSAGE	specifies full or concise message format.
	MAP	lists offsets of variables in static control section and DSAs.
	OFFSET	lists statement numbers associated with offsets.
	OPTIONS	lists options used.
	SOURCE	lists source program or preprocessor output.
	STORAGE	lists storage used.
	XREF	lists statements in which each identifier is used.
Improve readability of source listing	**MARGINI**	highlights any source outside margins.
	NEST	indicates do-group and block level by numbering in margin.
Control lines per page	**LINECOUNT**	specifies number of lines per page on listing.
Input options	**GRAPHIC**	specifies that shift codes can be used in source.
	MARGINS	identifies position of PL/I source and a carriage control character.
	NOT	used to specify up to seven alternate symbols for the logical NOT operator.
	OR	used to specify up to seven alternate symbols for the logical OR operator and the string concatenation operator.
	SEQUENCE	specifies the columns used for sequence numbers.
Options to prevent unnecessary processing	**COMPILE**	stops processing after errors are found in syntax checking.
	SYNTAX	stops processing after errors are found in preprocessing.
Options for preprocessing	**INCLUDE**	allows secondary input to be included without using preprocessor.
	MACRO	allows preprocessor to be used.
	MDECK	produces a source deck from preprocessor output.
Option to improve performance	**OPTIMIZE**	improves run-time performance or specifies faster compile time.
Options to use when producing an object module	**NAME**	specifies the object module will be given a specific external name.
	SYSTEM	specifies the parameter list format that is passed to the main procedure.
Option to control storage	**SIZE**	controls the amount of storage used by the compiler.
Options to specify statement numbering system	**NUMBER & GONUMBER**	numbers statements according to line in which they start.
	STMT & GOSTMT	numbers statements sequentially.
Option to control compile-time options	**CONTROL**	specifies that any compile-time options previously deleted are available.
Option to control language level	**LANGLVL**	defines the level of language supported.
Option to produce program structure information	**TEST**	defines the hooks and breakpoints that will be included in the object code.

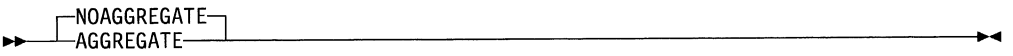
Table 4 (Page 2 of 2). Compile-time options arranged by function

Options ignored by PL/I VSE but retained for compatibility with other implementations	**CMPAT**	controls level of compatibility with previous releases.
	DECK	produces an object module in punched card format.
	IMPRECISE	allows imprecise interrupts to be handled correctly.
	INTERRUPT	specifies that the ATTENTION condition will be raised after an interrupt occurs.
	OBJECT	produces an object module from compiled output.
	TERMINAL	specifies how much of listing is transmitted to terminal.

Compile-time option descriptions

AGGREGATE

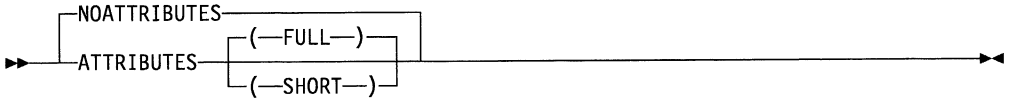
The AGGREGATE option specifies that the compiler includes an aggregate-length table that gives the lengths of all arrays and major structures in the source program in the compiler listing.



Abbreviations: AG for AGGREGATE
NOAG for NOAGGREGATE

ATTRIBUTES

The ATTRIBUTES option specifies that the compiler includes a table of source-program identifiers and their attributes in the compiler listing. If you include both ATTRIBUTES and XREF, 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.



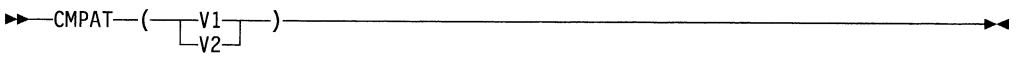
Abbreviations: A for ATTRIBUTES
NA for NOATTRIBUTES
F for FULL
S for SHORT

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.

CMPAT

The CMPAT option has no effect in VSE. It is syntax-checked and ignored.



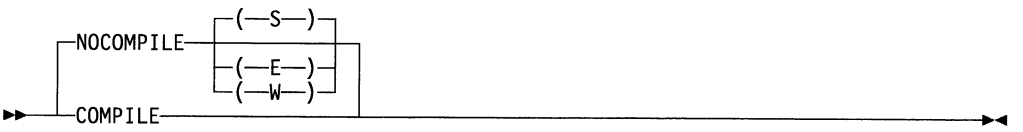
Abbreviation: CMP for CMPAT

This option is used in MVS and VM to define the level of compatibility with previous releases of PL/I. In VSE, compatibility with DOS PL/I Release 6 is maintained only at the source code level. Object code compatibility is not supported by this release of PL/I. For information on migrating from DOS PL/I Release 6 to PL/I VSE, see *PL/I VSE Migration Guide.*

COMPILE

The COMPILE option specifies that the compiler compiles the source program unless it detects an unrecoverable error during preprocessing or syntax checking. Link-edit will continue unless an unrecoverable error is detected during the compile.

With the NOCOMPILE option, whether the compiler continues after syntax checking depends on the severity of the error detected, as described in the list below. Link-edit will only continue if the severity is less than severe, error, or warning, depending on the option specified (see the list below).



Abbreviations: C for COMPILE
NC for NOCOMPILE

NOCOMPILE
No compilation, even if no errors are detected. Processing stops unconditionally after syntax checking.

NOCOMPILE(S)
No compilation if a severe error or unrecoverable error is detected. Link-edit will only continue if the severity is less than severe.

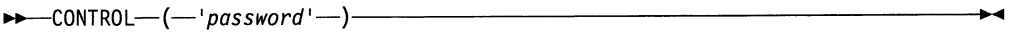
NOCOMPILE(E)
No compilation if an error, severe error, or unrecoverable error is detected. Link-edit will only continue if the severity is less than error.

NOCOMPILE(W)
No compilation if a warning, error, severe error, or unrecoverable error is detected. Link-edit will only continue if the severity is less than warning.

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

CONTROL

The CONTROL option specifies that any compile-time options deleted for your installation are available for this compilation. Using the CONTROL option alone does not restore compile-time options you have deleted from your system. You still must specify the appropriate keywords to use the options. The CONTROL option must be specified with a password that is established for each installation. If you use an incorrect password, processing will be terminated. If you use the CONTROL option, it must be specified first in the list of options.

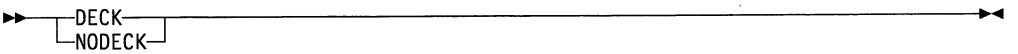


password
is a character string not exceeding eight characters.

DECK

DECK/NODECK should not be specified as a compiler option. If you specify it, it will be ignored.

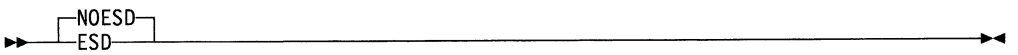
To adhere to VSE/ESA conventions, DECK/NODECK should only be specified via the JCL OPTION statement. See "Job control for compilation" on page 44 for details.



Abbreviations: D for DECK
ND for NODECK

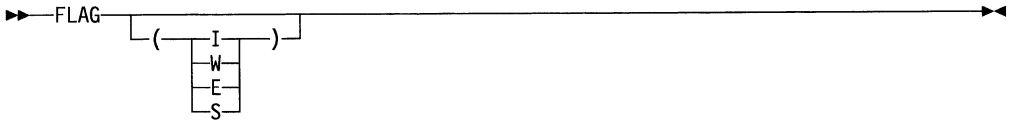
ESD

The ESD option specifies that the external symbol dictionary (ESD) is listed in the compiler listing.



FLAG

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



Abbreviation: F for FLAG

FLAG(I)
List all messages.

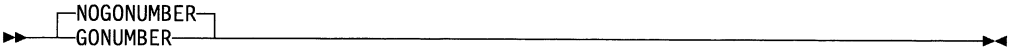
FLAG(W)
List all except information messages. If you specify FLAG, FLAG(W) is assumed.

FLAG(E)
List all except warning and information messages.

FLAG(S)
List only severe error and unrecoverable error messages.

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.



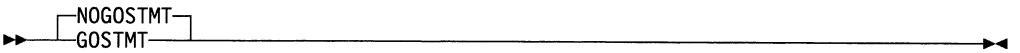
Abbreviations: GN for GONUMBER
NGN for NOGONUMBER

Alternatively, these line numbers can be derived by using the offset address, which is always included in run-time messages, and the table produced by the OFFSET option. (The NUMBER option must also apply.)

The GONUMBER option implies NUMBER, NOSTMT, and NOGOSTMT. The OFFSET option is separate from these numbering options and must be specified if required.

GOSTMT

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



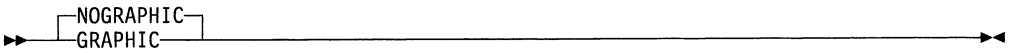
Abbreviations: GS for GOSTMT
NGS for NOGOSTMT

These statement numbers can also be derived by using the offset address, which is always included in run-time messages, and the table produced by the OFFSET option. (The STMT option must also apply.)

The GOSTMT option implies STMT, NONUMBER, and NOGONUMBER. The OFFSET option is separate from these numbering options and must be specified if required.

GRAPHIC

Using 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. This includes occurrences in comments and string constants.



Abbreviations: GR for GRAPHIC
NGR for NOGRAPHIC

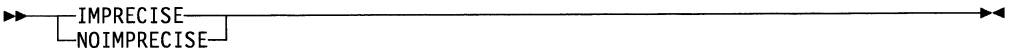
The GRAPHIC compile-time option must be specified if the source program uses any of the following:

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

See the description of the ATTRIBUTE and cross-reference table on page 35 for a discussion on the ordering of DBCS identifiers.

IMPRECISE

The IMPRECISE option has no effect in VSE. It is syntax-checked and ignored.

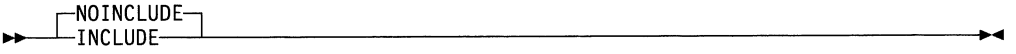


Abbreviations: IMP for IMPRECISE
NIMP for NOIMPRECISE

The IMPRECISE option is kept for compatibility with the MVS and VM version of the compiler.

INCLUDE

The INCLUDE option specifies that %INCLUDE statements are handled without using the full preprocessor facilities and incurring more overhead. This method is faster than using the preprocessor for programs that use the %INCLUDE statement but no other PL/I preprocessor statements. The INCLUDE option has no effect if preprocessor statements other than %INCLUDE are used in the program. In these cases, the MACRO option must be used.

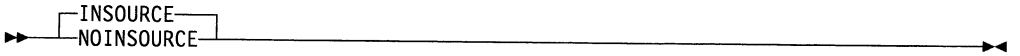


Abbreviations: INC for INCLUDE
NINC for NOINCLUDE

If you specify the MACRO option, it overrides the INCLUDE option.

INSOURCE

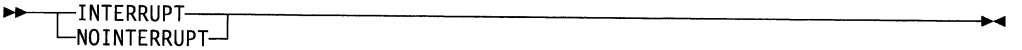
The INSOURCE option specifies that the compiler should include a listing of the source program before the PL/I macro preprocessor translates it. Thus, the INSOURCE listing contains preprocessor statements that do not appear in the SOURCE listing. This option is applicable only when the MACRO option is in effect.



Abbreviations: IS for INSOURCE
NIS for NOINSOURCE

INTERRUPT

The INTERRUPT option is not applicable for VSE/ESA. If specified, it will be syntax checked and ignored.

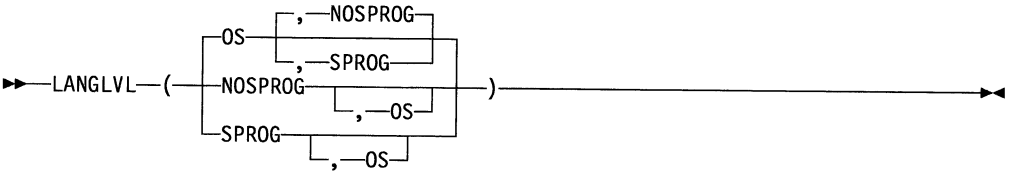


Abbreviations: INT for INTERRUPT
NINT for NOINTERRUPT

The INTERRUPT option is kept for compatibility with the MVS and VM implementation of the compiler. It determines the effect of attention interrupts when the compiled PL/I program runs under an interactive system such as CMS or TSO.

LANGLVL

The LANGLVL option specifies the level of PL/I language supported, including whether pointers in expressions are to be supported.



OS

specifies the level of PL/I language the compiler is to support. LANGLVL(OS) is always in effect. It specifies that the compiled code can run as an application program under the current operating system (VSE).

NOSPROG

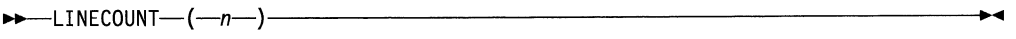
specifies that the compiler is ***not*** to allow the additional support for pointers allowed under SPROG.

SPROG
specifies that the compiler is to allow extended operations on pointers, including arithmetic, and the use of the POINTERADD, BINARYVALUE, and POINTERVALUE built-in functions.

For more information on pointer operations, see the *PL/I VSE Language Reference* book.

LINECOUNT

The LINECOUNT option specifies the number of lines included in each page of the compiler listing, including heading lines and blank lines.

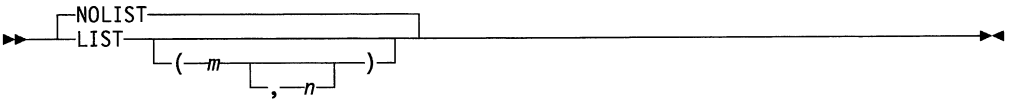


Abbreviation: LC for LINECOUNT

n is the number of lines. It must be in the range 1 through 32,767, but only headings are generated if you specify less than 7. When you specify less than 100, the static internal storage map and the object listing are printed in double column format. Otherwise, they are printed in single column format.

LIST

The LIST option specifies that the compiler includes a listing of the object module (in a syntax similar to assembler language instructions) in the compiler listing. If both *m* and *n* are omitted, the compiler produces a listing of the whole program.



m is the number of the first, or only, source statement for which an object listing is required.

n is the number of the last source statement for which an object listing is required. If *n* is omitted, only statement *m* is listed.

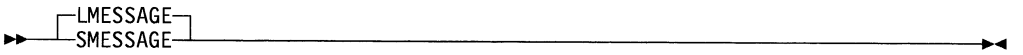
If the option NUMBER applies, *m* and *n* must be specified as line numbers. If the STMT option applies, *m* and *n* must be statement numbers.

If you use LIST in conjunction with MAP, it increases the information generated by MAP. (See "MAP" on page 14 for more information on the MAP compile-time option.)

If the LINECOUNT option specifies a value less than 100, the object listing is printed in double-column format. If LINECOUNT is 100 or more, the listing is in single-column format, which is easier for viewing at a terminal.

LMESSAGE

The LMESSAGE and SMESSAGE options specify the format of messages produced by the compiler.

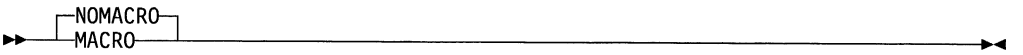


Abbreviations: LMSG for LMESSAGE
SMSG for SMESSAGE

LMESSAGE specifies that compiler messages will be produced in long form, and SMESSAGE specifies that the messages will be in short form.

MACRO

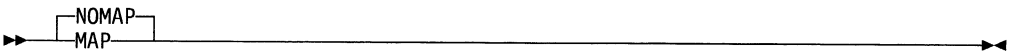
The MACRO option specifies that the source program will be processed by the preprocessor. MACRO overrides INCLUDE if both are specified.



Abbreviations: M for MACRO
NM for NOMACRO

MAP

The MAP option specifies that the compiler produces tables showing the organization of the static storage for the object module. These tables show how variables are mapped in the static internal control section and in DSAs, thus enabling STATIC INTERNAL and AUTOMATIC variables to be found in PLIDUMP. If LIST (described under "LIST" on page 13) is also specified, the MAP option produces tables showing constants, control blocks and INITIAL variable values. LIST generates a listing of the object code in pseudo-assembler language format.



If you want a complete map, but not a complete list, you can specify a single statement as an argument for LIST to minimize the size of the LIST. For example:

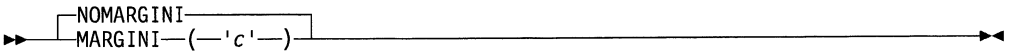
```
%PROCESS MAP LIST(1);
```

If the LINECOUNT option specifies a value less than 100, the map is printed in double-column format. If LINECOUNT is 100 or more, the map listing is in single-column format, which is easier for viewing at a terminal.

MARGINI

The MARGINI option specifies that the compiler includes a specified character in the column preceding the left-hand margin, and also in the column following the right-hand margin of the listings that the compiler produces when you use 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.



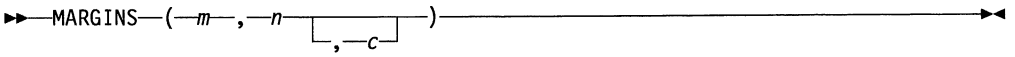
Abbreviations: MI for MARGINI
NMI for NOMARGINI

c is the character to be printed as the margin indicator.

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.



Abbreviation: MAR for MARGINS

m is the column number of the leftmost character (first data byte) that is processed by the compiler. It must not exceed 80.

n is the column number of the rightmost character (last data byte) that is processed by the compiler. It must be greater than *m*, but not greater than 80.

c is the column number of the ANS printer control character. It must not exceed 80 and must 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 situation causes the format of the listing to be unpredictable. To avoid this problem, put the carriage control character to the left of the source margins.

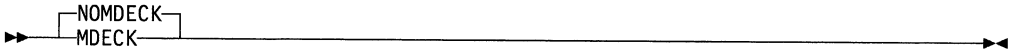
Specifying MARGINS(,,c) is an alternative to using %PAGE and %SKIP statements (described in the *PL/I VSE Language Reference).*

The IBM-supplied default is MARGINS(2,72). 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.

MDECK

The MDECK option specifies that the preprocessor produces a copy of its output on SYSPCH. The MDECK option allows you to retain the output from the preprocessor as a file of 80-column records.

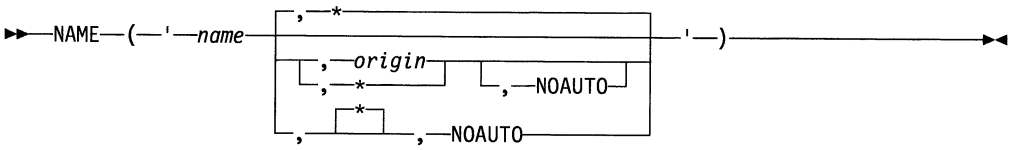


Abbreviations: MD for MDECK
NMD for NOMDECK

The compiler ignores MDECK if NOMACRO is in effect.

NAME

The NAME option specifies that the object deck created by the compiler is given the specified external name.



Abbreviation: N for NAME

name
has from one through eight characters, and begins with an alphabetic character. NAME has no default.

origin
is the same as the *origin* parameter that is used by the VSE/ESA Linkage Editor in the PHASE statement. It defaults to `'*'`.

NOAUTO
is optional, and will be copied to the linkage editor PHASE statement if entered. See Chapter 4, "Link-editing and running" on page 58 for more information on the linkage editor.

When used in conjunction with the JCL option LINK or CATAL, the NAME option instructs the compiler to insert a Linkage Editor PHASE statement in front of the object code on SYSLNK. The format of the PHASE statement is:

```
PHASE name,*
```

where `'name'` is the name of the object module derived from the NAME option.

When used in conjunction with the JCL option DECK, the NAME option instructs the compiler to produce a VSE Librarian CATALOG command card in front of the object module on SYSPCH. The format of the CATALOG command is:

```
CATALOG name.OBJ,REPLACE=YES
```

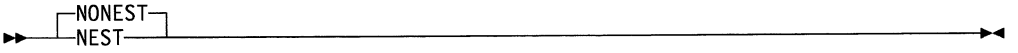
where 'name' is the name of the object module derived from the NAME compiler option.

If you do not specify any of the DECK, LINK or CATAL options in the JCL OPTION statement, the the compiler will ignore the NAME option if it is coded.

If you specify both of the JCL options DECK and LINK (or CATAL), then the compiler will generate both a PHASE statement in the SYSLNK output, and a CATALOG statement in the SYSPCH output.

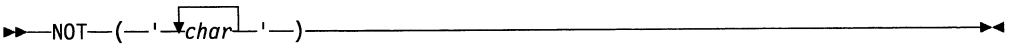
NEST

You can use the NEST option to specify that the listing resulting from the SOURCE option indicates the block level and the do-group level for each statement.



NOT

The NOT option specifies up to seven alternate symbols, any one of which can be used as the logical NOT operator.



char

is a single SBCS character.

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

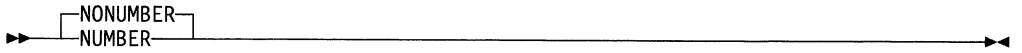
If you specify the NOT option, the standard NOT symbol is no longer recognized unless you specify it as one of the characters in the character string.

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

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

NUMBER

The NUMBER option specifies that numbers in the sequence fields in the source input records are used to derive the statement numbers in the listings resulting from the AGGREGATE, ATTRIBUTES, LIST, OFFSET, SOURCE and XREF options.



Abbreviations: NUM for NUMBER
NNUM for NONUMBER

You can specify the position of the sequence field in the SEQUENCE option. Otherwise the compiler assumes that the sequence number is contained in the last eight columns of the source input records (columns 73-80).

Note: The preprocessor will always put sequence numbers into columns 73-80 of its output records.

The compiler calculates the line number from the five right-hand characters of the sequence number (or the number specified, if less than five). These characters are converted to decimal digits if necessary. Each time the compiler finds a line number that is not greater than the preceding line number, it forms a new line number by adding the minimum integral multiple of 100,000 to produce a line number that is greater than the preceding one. The compiler issues a message to warn you of the adjustment, except when you specify the INCLUDE option or the MACRO option.

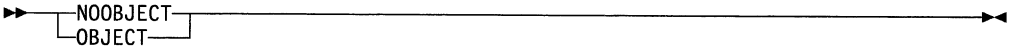
If there is more than one statement on a line, the compiler uses a suffix to identify the actual statement in the messages. For example, the second statement beginning on the line numbered 40 is identified by the number 40.2. The maximum value for this suffix is 31. Thus the thirty-first and subsequent statements on a line have the same number.

If the sequence field consists only of blanks, the compiler forms the new line number by adding 10 to the preceding one. The maximum line number allowed by the compiler is 134,000,000. Numbers that would normally exceed this are set to this maximum value. Only eight digits print in the source listing; line numbers of 100,000,000 or over print without the leading *1* digit.

If you specify NONUMBER, STMT and NOGONUMBER are implied. NUMBER is implied by NOSTMT or GONUMBER.

OBJECT

OBJECT/NOOBJECT should not be specified as a compiler option. If you specify it, it will be ignored.

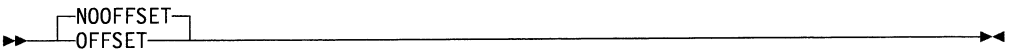


Abbreviations: OBJ for OBJECT
NOBJ for NOOBJECT

To adhere to VSE/ESA conventions, the compiler examines the options specified in the JCL OPTION statement (OPTION LINK or OPTION CATAL) to determine whether or not to produce an object code file. See "Job control for compilation" on page 44 for details.

OFFSET

The OFFSET option specifies that the compiler prints a table of statement or line numbers for each procedure with their offset addresses relative to the primary entry point of the procedure. You can use this table to identify a statement from a run-time error message if the GONUMBER or GOSTMT option is not in effect.



Abbreviations: OF for OFFSET
NOF for NOOFFSET

If GOSTMT applies, the run-time library includes statement numbers, as well as offset addresses, in run-time messages. If GONUMBER applies, the run-time library includes line numbers, as well as offset addresses, in run-time messages.

For more information on determining line numbers from the offsets given in error messages, see "Statement offset addresses" on page 38.

OPTIMIZE

The OPTIMIZE option specifies the type of optimization required:

Abbreviations: OPT for OPTIMIZE
NOPT for NOOPTIMIZE

OPTIMIZE(TIME)
specifies that the compiler 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. During optimization the compiler can move code to increase run-time efficiency. As a result, statement numbers in the program listing cannot correspond to the statement numbers used in run-time messages.

OPTIMIZE(0)
is the equivalent of NOOPTIMIZE.

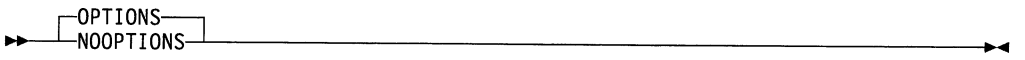
OPTIMIZE(2)
is the equivalent of OPTIMIZE(TIME).

NOOPTIMIZE
specifies fast compilation speed, but inhibits optimization.

For a full discussion of optimization, see Chapter 10, "Efficient programming" on page 220.

OPTIONS

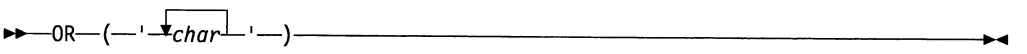
The OPTIONS option specifies that the compiler includes a list showing the compile-time options to be used during this compilation in the compiler listing. This list includes all options applied by default, those specified in the JCL for the compilation, and those specified in a %PROCESS statement.



Abbreviations: OP for OPTIONS
NOP for NOOPTIONS

OR

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



char

is a single SBCS character.

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

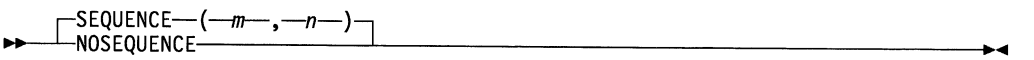
For example, OR('\') means that the backslash character, X'E0', will be recognized as the logical OR operator, and two consecutive backslashes will be recognized as the concatenation operator. The standard OR symbol, '|', X'4F', will not be recognized as either operator. Similarly, OR('\|') means that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

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

SEQUENCE

The SEQUENCE option defines the section of the input record from which the compiler takes the sequence numbers. These numbers are included in the source listings produced by the INSOURCE and SOURCE option.

The compiler uses sequence numbers to calculate statement numbers if the NUMBER option is in effect. The compiler does not sort the input lines or records into the specified sequence.



Abbreviations: SEQ for SEQUENCE
NSEQ for NOSEQUENCE

m specifies the column number of the left-hand margin.

n specifies the column number of the right-hand margin.

The extent specified should not overlap with the source program (as specified in the MARGINS option).

The IBM-supplied default is SEQUENCE(73,80).

If the SEQUENCE option is used, an external procedure cannot contain more than 32,767 lines. To Compile an external procedure containing more than 32,767 lines, you must specify the NOSEQUENCE option. You should also not specify the NUMBER or GONUMBER options, because these imply SEQUENCE.

Note: The preprocessor will always produce output with sequence numbers in columns 73-80, regardless of your SEQUENCE option.

SIZE

You can use this option to limit the amount of GETVIS storage the compiler uses for additional work space. This is of value, for example, when dynamically invoking the compiler, to ensure that space is left for other purposes. There are five forms of the SIZE option:

Abbreviation: SZ for SIZE

SIZE(yyyyyyyy)
specifies that yyyyyyyy bytes of GETVIS storage are requested. Leading zeros are not required.

SIZE(yyyyyK)
specifies that yyyyyK bytes of GETVIS storage are requested (1K=1024). Leading zeros are not required.

SIZE(MAX)
specifies that the compiler obtains as much GETVIS storage as it can, and then releases 55K for operating system overheads.

SIZE(-yyyyyy)
specifies that the compiler obtains as much GETVIS storage as it can, and then releases yyyyyy bytes to the operating system. Leading zeros are not required.

SIZE(-yyyK)
specifies that the compiler obtains as much GETVIS storage as it can, and then releases yyyK bytes to the operating system (1K=1024). Leading zeros are not required.

The IBM-supplied default is SIZE(MAX), which instructs the compiler to obtain the largest available contiguous block of GETVIS storage, minus 55K for operating system overheads.

Note that with the *yyyy* forms of SIZE, the compiler does not make any adjustments for overheads. It simply obtains the specified amount of storage.

The negative forms of SIZE can be useful when a certain amount of space must be left free and the maximum size is unknown, or can vary because the job is run in partitions of different sizes.

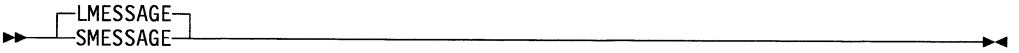
When a limit is specified, the amount of storage used by the compiler depends on how the operating system has been generated, and the method used for storage allocation. The compiler assumes that buffers, data management routines, and processing phases take up a fixed amount of main storage, but this amount can vary unknown to the compiler.

After the compiler has loaded its initial phases and opened all files, it attempts to allocate space for working storage. If a limit is specified, then this amount is requested. If the amount available is less than specified, but is more than the minimum workspace required, compilation proceeds. If insufficient storage is available, compilation is terminated. This latter situation should arise only if too much space for buffers has been requested or if the partition is too small. The value you specify in the SIZE option cannot exceed the main storage available for the job step and cannot be changed after processing has begun.

This means that in a batched compilation the value you establish when you invoke the compiler cannot be changed for later programs in the batch. Thus it is ignored if specified in a %PROCESS statement other than the first in batched compilations.

SMESSAGE

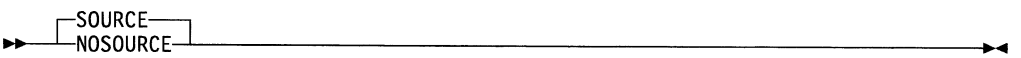
The LMESSAGE and SMESSAGE options specify the format of messages produced by the compiler.



LMESSAGE specifies that compiler messages will be produced in long form, and SMESSAGE specifies that the messages will be in short form.

SOURCE

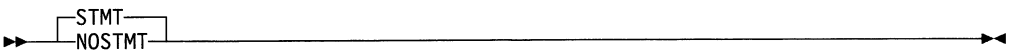
The SOURCE option specifies that the compiler includes a listing of the source program in the compiler listing. The source program listed is either the original source input or, if the MACRO option applies, the output from the preprocessor.



Abbreviations: S for SOURCE
NS for NOSOURCE

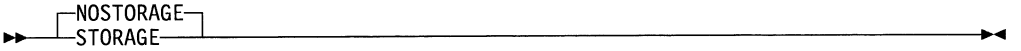
STMT

The STMT option specifies that statements in the source program are counted, and this statement number is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, LIST, OFFSET, SOURCE, and XREF options. STMT is implied by NONUMBER or GOSTMT. If NOSTMT is specified, NUMBER and NOGOSTMT are implied.



STORAGE

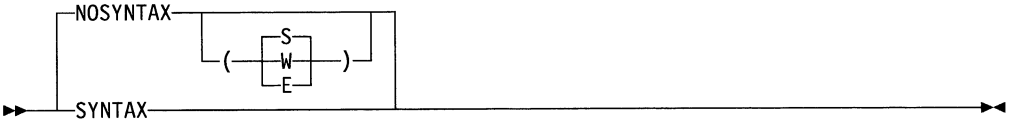
The STORAGE option specifies that the compiler includes a table giving the main storage requirements for the object module in the compiler listing.



Abbreviations: STG for STORAGE
NSTG for NOSTORAGE

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.



Abbreviations: SYN for SYNTAX
NSYN for NOSYNTAX

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.

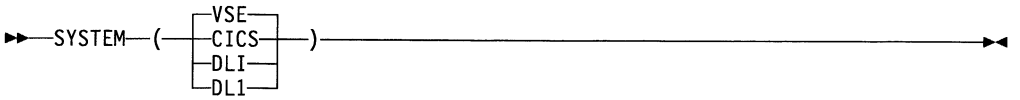
If the SOURCE option applies, the compiler generates a source listing even if it does not perform syntax checking.

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

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

SYSTEM

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



SYSTEM(VSE)
Specifies that the parameter list passed to the program is in the form of a single varying character string, or there are no parameters. The program will run as a VSE batch program.

SYSTEM(CICS)
Specifies that the parameter list passed to the program is in the form of a pointer or list of pointers. The program will run as a CICS transaction.

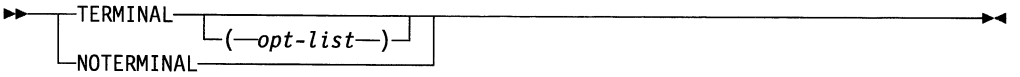
SYSTEM(DLI)
Specifies that the parameter list passed to the program is in the form of a pointer or list of pointers. The program will run as a VSE batch program.

SYSTEM(DL1)
See SYSTEM(DLI).

For more information, see *LE/VSE Programming Guide.*

TERMINAL

The TERMINAL option has no effect in a VSE environment. If specified, it will be syntax-checked and ignored.

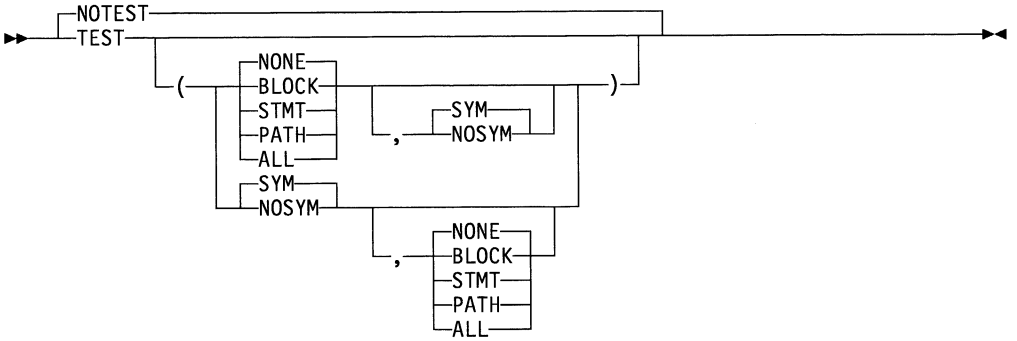


Abbreviations: TERM for TERMINAL
NTERM for NOTERMINAL

This option is retained for compatibility with the MVS and VM version of the compiler. It is applicable only in a conversational environment, such as TSO or CMS.

TEST

The TEST option specifies the level of testing capability that the compiler generates as part of the object code. This option tells the compiler to insert hooks at the specified points in the code, which can then be used for program debugging and tuning. See Chapter 9, "Examining and tuning compiled modules" on page 181 for a discussion of how these hooks can be used to report on the behavior and performance of PL/I programs.



The TEST option can imply GONUMBER or GOSTMT, depending on whether NUMBER or STMT is in effect.

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.

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

STMT
Specifies that the compiler inserts hooks at statement boundaries and block boundaries. STMT causes a statement table to be generated.

PATH
tells the compiler to insert hooks:

- Before the first statement enclosed by an iterative DO statement
- Before the first statement of the true part of an IF statement
- Before the first statement of the false part of an IF statement
- Before the first statement of a true WHEN or OTHERWISE statement of a SELECT group
- Before the statement following a user label
- 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.

ALL
tells the compiler to insert hooks at all possible locations and to generate a statement table.

NONE
tells the compiler not to put hooks into the program.

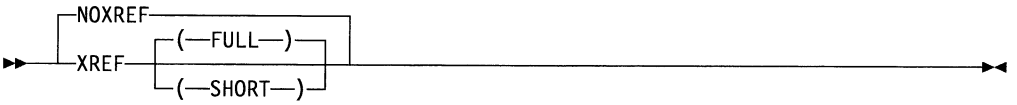
SYM
tells the compiler to create a symbol table that will allow you to examine variables by name.

NOSYM
tells the compiler not to generate a symbol table.

NOTEST
suppresses the generation of all testing information.

XREF

The XREF option specifies that the compiler includes 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. (The only exception is that label references on END statements are not included. 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.)



Abbreviations: X for XREF
NX for NOXREF
F for FULL
S for SHORT

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

SHORT
Unreferenced identifiers are omitted from the compiler listing.

For a description of the format and content of the cross-reference table, see "Cross-reference table" on page 35.

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.

Specifying compiler options

You can specify options for the compiler in the JCL that you use to compile your program, or in a %PROCESS or *PROCESS statement. The %PROCESS statement is described in the following section, and the JCL option is described in "Specifying compiler options" on page 47.

If you specify conflicting options (for example, NOMAP and MAP), the compiler uses the last specification. The order of precedence (from highest to lowest) for specifying compiler options is:

1. The %PROCESS or *PROCESS statement in your source program,
2. Your compile JCL (the PARM parameter of the // EXEC IEL1AA statement),
3. The installation defaults. These are the default options that were defined when the compiler was installed on your system.

Specifying options in the %PROCESS or *PROCESS statements

The compiler uses the %PROCESS statement to identify the start of each external procedure and to allow 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 or percent sign must appear in column 1 of the record, and 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 can split option keywords or keyword arguments when continuing onto the next record, provided that the keyword or argument string terminates in the right-hand source margin, and the remainder of the string starts in the same column as the asterisk or percent sign. 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 F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST SYSTEM(VSE) ;
```

For information about using the %PROCESS statement with batched compilation, see "Compiling multiple procedures in a single job step" on page 48.

Compile-time options, their abbreviated syntax, and their IBM-supplied defaults are shown in Table 3 on page 4 and Table 4 on page 6. Your site may have changed the IBM-supplied defaults or deleted options. Be sure to check for any changes before using compile-time option defaults. You can reinstate deleted compile-time options for a compilation by using the CONTROL compile-time option.

Using the preprocessor

The preprocessing facilities of the compiler are described in the *PL/I VSE Language Reference* book. 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 of the use of the preprocessor and explains how to establish and use source statement libraries.

Invoking the preprocessor

If you specify the compile-time option MACRO, the preprocessor stage of the compiler is executed. The compiler and the preprocessor use the work file IJSYS01 during processing. They also use this work file to store the preprocessed source program until compilation begins.

The format of the preprocessor output as it appears in the compiler source listing is given in Table 5.

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, then additional lines are included for any lines that need continuation. If the original source program used fewer than 71 columns, then extra blanks are added on the right.
Columns 73-80	Sequence number, right-aligned. If either SEQUENCE or NUMBER applies, this is taken from the sequence number field. Otherwise, it is a preprocessor generated number, in the range 1 through 99999. This sequence number will be used in the listing produced by the INSOURCE and SOURCE options, and in any preprocessor diagnostic messages.
Column 81	blank
Columns 82, 83	Two-digit number giving the maximum depth of replacement by the preprocessor for this line. If no replacement occurs, the columns are blank.
Column 84	*E* signifying that an error occurred while replacement was being attempted. If no error occurred, the column is blank.

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 16, INSOURCE on page 12, and SYNTAX on page 23.

A simple example of the use of the preprocessor to produce a source deck is shown in Figure 1 on page 29. According to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine (CITYSUB) or a function (CITYFUN). The preprocessor output will be placed on a data set, which can then be used as input to the VSE Librarian to put the resultant program into a source library. Normally compilation would continue and the preprocessor output would be compiled.

```
// JOB    OPT4#8
// DLBL   IJSYSPH,'COMMON.UTILS.FUN',0,SD
// EXTENT SYSPCH,VSE222,1,0,3450,5
ASSGN     SYSPCH,DISK,VOL=VSE222,SHR
// EXEC   IEL1AA,SIZE=128K
%PROCESS MACRO,MDECK,NOCOMPILE,NOSYNTAX;
 /* 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;
/*
CLOSE     SYSPCH,PUNCH
/&
```

Figure 1. Using the preprocessor to produce a source deck that can be placed in a source program library

Using the %INCLUDE statement

The *PL/I VSE Language Reference* describes how to use the %INCLUDE statement to incorporate source text from a library into a PL/I program. A *library* is a file system managed by the VSE Librarian, that can be used to store source code as library members. The Librarian format for a PL/I source member name is 'name.type' where 'name' is the 1-8 character member name and 'type' is a single character that defaults to 'P'.

Source text that you may want to insert into a PL/I program using a %INCLUDE statement must exist as a member within a library. "Compiler data sets" on page 45 further describes the process of defining a source statement library to the compiler.

The statement:

```
%INCLUDE P(INVERT);
```

or simply

```
%INCLUDE INVERT;
```

specifies that the source statements in member INVERT.P in a library are to be inserted consecutively into the source program. The compilation JCL must include a LIBDEF SOURCE search chain, which specifies the names of the libraries to search for the specified member.

Included members can themselves contain %INCLUDE statements, providing a nesting facility. There is no limit to the level of nesting available when the MACRO compiler option is used, but when the INCLUDE option is used the nesting level is limited to 8.

Note: For compatibility with the VM and MVS implementations of PL/I, the *type* specification on the %INCLUDE statement can contain up to 8 characters, for example:

```
%INCLUDE SYSLIB(LAYOUT);
```

If you specify a *type* of more than one character, the compiler issues an informational error message, and assumes the default member type of `'P'`.

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 COMMON.UTILS is defined in the LIBDEF SOURCE search sequence in the compile JCL.

It is not necessary to invoke the preprocessor if your source program, and any text to be included, does not contain any macro statements. Under these circumstances, you can obtain faster inclusion of text by specifying the INCLUDE compile-time option.

```
// JOB    OPT4#9
// OPTION LINK
// LIBDEF SOURCE,SEARCH=(COMMON.UTILS)
// EXEC   IEL1AA,SIZE=128K
%PROCESS INCLUDE;
   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                        */
/*
// EXEC   LNKEDT
// EXEC   ,SIZE=128K
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. These statements must not have label or condition prefixes, and cannot be a "unit" of a compound statement.

The % statements allow you to control the source program listing and to include external strings in the source program. These control statements, %INCLUDE, %PRINT, %NOPRINT, %PAGE, and %SKIP, are listed below and described fully in the *PL/I VSE 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.

%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.

%SKIP — Specifies the number of lines to be skipped.

Note: You should place each % statement on a line by itself.

Invoking the compiler from an assembler routine

You can invoke the compiler from an Assembler language program by using the CDLOAD macro instruction to load the compiler's primary phase, and then the ATTACH or CALL macro instruction to pass control to the compiler. The following information supplements the description of these macro instructions given in the *VSE/ESA System Macros Reference* book.

To load the compiler specify IEL1AA as the phase name. When the compiler finishes it will return to your program with a return code in register 15 indicating its level of success or failure. The meanings of the return codes are described in "Return codes" on page 43.

You can pass three address parameters to the compiler:

1. The address of a compile-time option list
2. A zero address (this is a placeholder for compatibility with the MVS implementation)
3. The address of a page number that is to be used for the first page of the compiler listing on SYSLST

These addresses must be in adjacent fullwords, aligned on a fullword boundary. Register 1 must point to the first address in the list, and the first (left-hand) bit of the last address must be set to 1, to indicate the end of the list.

The format of the three parameters is described below.

Option list

The option list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). The

remainder of the list can comprise any of the compile-time option keywords, separated by one or more blanks, a comma, or both of these.

Zero address
The second parameter points to a DDNAME list in the MVS implementation of the compiler. It is not used under VSE, and should be set to zero.

Page number
The page number parameter must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the page number field. This must be either zero or four. If it is zero, the compiler will start numbering pages from 1. If it is 4, the following four bytes must contain a fullword binary number, which the compiler will use as the starting page number for the listing. The compiler will then add 1 to the last page number used in the listing and put this value back in the page-number field before returning control to the invoking routine. Thus, if the compiler is invoked again, page numbering is continuous.

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. It places this listing in the data set on SYSLST. The following description of the listing refers to its appearance on a printed page.

The first part of Table 4 on page 6 shows the components that can be included in the compiler listing. The rest of this section describes them in detail.

The listing comprises a small amount of standard information that always appears, together with those items of optional information specified or supplied by default.

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" on page 42. The penultimate 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, then the external procedure name "xxxx" is not included and the line truncates to "END OF COMPILATION."

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

Options used for the compilation

If the option OPTIONS applies, a complete list of the options specified for the compilation, including the default options, appears on the first page.

Preprocessor input

If both the options MACRO and INSOURCE apply, 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" on page 42.

Source program

If the option SOURCE applies, the input to the compiler is listed, one record per line. If the input records contain printer control characters or %SKIP or %PAGE statements, the lines are spaced accordingly. You can use %NOPRINT and %PRINT statements to stop and restart the printing of the listing.

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

If the MACRO option does not apply but the INCLUDE option does, any included text is bracketed by comments indicating the %INCLUDE statement that caused the text to be included.

Assume the following source input on SYSIPT:

```
MAIN: PROC REORDER;
%INCLUDE MEMBER1;
END;
```

and the following content of MEMBER1:

```
J=K;
%INCLUDE DECLARES;
L=M;
```

and the following content of DECLARES:

```
DCL (NULL,DATE) BUILTIN;
```

The resultant source listing if the INCLUDE option is used will be:

```
MAIN: PROC REORDER;
/*BEGIN %INCLUDE P(MEMBER1)*****************************************/
J=K;
/*BEGIN %INCLUDE P(DECLARES)****************************************/
DCL (NULL,DATE) BUILTIN;
/*END   %INCLUDE P(DECLARES)****************************************/
L=M;
/*END   %INCLUDE P(MEMBER1)*****************************************/
END;
```

If the STMT compile-time option applies, the statement numbers are derived from a count of the number of statements in the program after %INCLUDEs have been processed.

If the NUMBER option applies, the compiler derives statement numbers from the sequence numbers of the statements in the source records after %INCLUDE statements have been processed. Normally the compiler uses the last five digits as statement numbers. If, however, this does not produce a progression of statements with successively higher numbers, the compiler adds 100000 to all statement numbers starting from the one that would otherwise be equal to or less than its predecessor.

For instance, if a primary input data set had the following lines:

```
A:PROC;                                                          00001000
%INCLUDE B;                                                      00002000
END;                                                             00003000
```

and member B contained:

```
C=D;                                                             00001000
E=F;                                                             00002000
G=H;                                                             00003000
```

then the source listing would be as follows:

```
SOURCE LISTING
    NUMBER

    1000   A:PROC;                                                        00001000
                /*BEGIN %INCLUDE P(B)*******************************************/00002000
  101000   C=D;                                                           00001000
  102000   E=F;                                                           00002000
  103000   G=H;                                                           00003000
                /*END   %INCLUDE P(B)*******************************************/
  203000   END;                                                           00003000
```

The additional 100000 has been introduced into the statement numbers at two points:

1. Beginning at the first statement of the included text (the statement C=D), and
2. Beginning with the first statement after the included text (the END statement).

If the source statements are generated by the preprocessor, columns 82-84 contain diagnostic information, as shown in Table 5 on page 28.

Statement nesting level

If the option NEST applies, 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, for example:

```
     STMT   LEV  NT

        1        0  A: PROC OPTIONS(MAIN);
        2    1   0  B: PROC;
        3    2   0     DCL K(10,10) FIXED BIN (15);
        4    2   0     DCL Y FIXED BIN (15) INIT (6);
        5    2   0     DO I=1 TO 10;
        6    2   1        DO J=1 TO 10;
        7    2   2           K(I,J) = N;
        8    2   2        END;
        9    2   1     BEGIN;
       10    3   1        K(1,1)=Y;
       11    3   1     END;
       12    2   1  END B;
       13    1   0  END A;
```

ATTRIBUTE and cross-reference table

If the option ATTRIBUTES applies, the compiler prints an attribute table containing a list of the identifiers in the source program together with their declared and default attributes. In this context, the attributes include any relevant options, such as REFER, and also descriptive comments, such as:

```
/*STRUCTURE*/
```

If the option XREF applies, the compiler prints a cross-reference table containing a list of the identifiers in the source program together with the numbers of the statements in which they appear. If both ATTRIBUTES and XREF apply, the two tables are combined. If the suboption SHORT applies, unreferenced identifiers are not listed.

If the GRAPHIC compile-time option is in effect, and at least one DBCS identifier is found in the compilation unit, then the DBCS identifiers will be placed ahead of the SBCS identifiers in the ATTRIBUTES and XREF listing. The DBCS identifiers are sorted in the sequence of their 16-bit binary code, and the SBCS identifiers are sorted in normal EBCDIC collating sequence.

Attribute table

If you declare an identifier explicitly, the compiler lists the number of the DECLARE statement. The compiler indicates an undeclared variable by asterisks. (The compiler also lists undeclared variables in error messages.) It also gives the statement numbers of statement labels and entry labels.

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 it replaces expressions with asterisks. Structure levels other than base elements also have their bounds replaced by asterisks.

For a character string or a bit string, the compiler prints the length, preceded by the word BIT or CHARACTER, as in the declaration, but it replaces an expression with an asterisk.

Cross-reference table

If you combine the cross-reference table with the attribute table, the numbers of the statements or lines in which a name appears follow the list of attributes for the name. The order in which the statement numbers appear is subject to any reordering of blocks that has occurred during compilation. In general, the compiler gives the statement numbers for the outermost block first, followed on the next line by the statement numbers for the inner blocks.

The compiler expands and optimizes PL/I text before it produces the cross-reference table. Consequently, some names that appear only once within a source statement can acquire multiple references to the same statement number. By the same token, other names can appear to have incomplete lists of references,

while still others can have references to statements in which the name does not appear explicitly.

For example:

- Duplicate references can be listed for items such as do-loop control variables, and for some aggregates.
- Optimization of certain operations on structures can result in incomplete listings in the cross-reference table. The numbers of statements in which these operations are performed on major or minor structures are listed against the names of the elements, instead of against the structure names.
- No references to PROCEDURE or ENTRY statements in which a name appears as a parameter are listed in the cross-reference table entry for that name.
- References within DECLARE statements to variables that are not being declared are not listed. For example, in the statements:

  ```
  DCL ARRAY(N);
  DCL STRING CHAR(N);
  ```

  no references to these statements would appear in the cross-reference table entry for N.
- The number of a statement in which an implicitly pointer-qualified based variable name appears is included not only in the list of statement numbers for that name, but also in the list of statement numbers for the pointer implicitly associated with it.
- The statement number of an END or LEAVE statement that refers to a label is not listed in the entry for the label.
- Automatic variables declared with the INITIAL attribute have a reference to the PROCEDURE or BEGIN statement for the block containing the declaration included in the list of statement numbers.

Aggregate length table

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

- The statement number in which the aggregate is declared.
- The name of the aggregate and the element within the aggregate.
- The level number of each item in a structure.
- The number of dimensions in an array.
- The byte offset of each element from the beginning of the aggregate. (The compiler does not give bit offsets for unaligned bit-string data). 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.

The table is completed with the sum of the lengths of all aggregates that do not contain adjustable elements.

The statement or line number identifies either the DECLARE statement for the aggregate, or, for a controlled aggregate, an ALLOCATE statement for the aggregate. An entry appears for each ALLOCATE statement involving a controlled aggregate, as such statements can have the effect of changing the length of the aggregate during run-time. Allocation of a based aggregate does not have this effect, and only one entry, which is that corresponding to the DECLARE statement, appears.

When passing an aggregate to a subroutine, the length of an aggregate might not be known during compilation, either because the aggregate contains elements having adjustable lengths or dimensions, or because the aggregate is dynamically defined. In these cases, the compiler prints the word *adjustable* or *defined* in the *offset* column and *param* for parameter in the *element length* and *total length* columns. Because the compiler might not know the length of an aggregate during compilation, it does not print padding information.

An entry for a COBOL mapped structure has the word COBOL appended. COBOL mapped structures are structures into which a program reads or writes a COBOL record, or a structure that can be passed between PL/I programs and COBOL programs. The COBOL entry appears if the compiler determines that the COBOL and PL/I mapping for the structure is different, and the creation of a temporary structure mapped according to COBOL synchronized structure rules is not suppressed by NOMAP, NOMAPIN, or NOMAPOUT.

If a COBOL entry does appear it is additional to the entry for the PL/I mapped version of the structure.

The compiler makes a separate entry in the aggregate table for every aggregate dummy argument or COBOL mapped structure.

Storage requirements

If the option STORAGE applies, the compiler lists the following information under the heading *Storage Requirements* on the page following the end of the aggregate length table:

- The length of the program control section. The program control section is the part of the object that contains the executable part of the program.
- The length of the static internal control section. This control section contains all storage for variables declared STATIC INTERNAL.
- The storage area in bytes for each procedure.
- The storage area in bytes for each begin block.
- The storage area in bytes for each ON-unit.
- The dynamic storage area in bytes for each procedure, begin block, and ON-unit. The dynamic storage area is acquired at activation of the block.

Statement offset addresses

If the option LIST applies, the compiler includes a pseudo-assembler listing in the compiler listing. You can use the offset given in run-time error messages to discover the erroneous statement, because the offsets in both run-time messages and the pseudo-assembler listing are relative to the start of the external procedure. Simply match the offset given in the error message with the offset in the listing to find the erroneous statement.

In the example shown in Figure 3, compile unit offset +17E occurs in the object listing under statement 6. Statement 6 is the erroneous statement.

```
          SOURCE LISTING

       1    M:PROC OPTIONS(MAIN);
       2      CALL A2;
       3      A1:PROC;
       4        N=3;
       5        A2:ENTRY;
       6          N=N/0;
       7      END;
       8    END;


-  OBJECT LISTING

 * STATEMENT NUMBER  6
 00016C  58 70 D 0C0              L     7,192(0,13)
 000170  48 60 3 02A              LH    6,42(0,3)
 000174  48 80 7 0B8              LH    8,N
 000178  1B 99                    SR    9,9
 00017A  8E 80 0 010              SRDA  8,16
 00017E  1D 86                    DR    8,6
 000180  12 99                    LTR   9,9
 000182  47 B0 2 02A              BNM   CL.13
 000186  5A 90 3 034              A     9,52(0,3)
 00018A                    CL.13  EQU   *
 00018A  8A 90 0 010              SRA   9,16
 00018E  40 90 7 0B8              STH   9,N

Message:

IBM0301S ONCODE=320  The ZERODIVIDE condition was raised.
         From compile unit M at entry point A2 at compile
         unit offset +0000017E at address 000201FE.
```

Figure 3. Finding statement number from a compile unit offset in an error message

If the OFFSET option applies, the compiler lists for each primary entry point the offsets at which statements occur. This information is found in the compiler listing under the heading, "Table of Offsets and Statement Numbers."

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.

If a secondary entry point is used, first find the name of the block that contains this entry and the corresponding section of the offset table that relates to this name. Next, add the offset given in the message to the offset of the secondary entry point in the table. This will convert the message offset so that it is relative to the primary entry point versus the secondary entry point, which was entered during execution.

Use this converted offset to search the section of the offset table for the largest offset as described above.

In the example in Figure 4, secondary entry point P2 is contained in procedure block P1 at offset X'78'. Adding X'78' to the message entry offset of X'44' yields a value of X'BC'. The largest offset table entry less than or equal to X'BC' is X'B4', which corresponds to statement number 7.

```
                SOURCE LISTING
   STMT

      1     Q: PROC OPTIONS(MAIN);
      2        ON ERROR SNAP GOTO L;
      3        CALL P2;
      4        P1: PROC;
      5            N=1;
      6        P2: ENTRY;
      7            SIGNAL ERROR;
      8            END;
      9        L: END;


           TABLE OF OFFSETS AND STATEMENT NUMBERS

                WITHIN PROCEDURE Q
   OFFSET (HEX)        0     A8     C0     CA
   STATEMENT NO.       1      2      3      9

                WITHIN PROCEDURE P1
   OFFSET (HEX)        0     78     A8     B4     BE
   STATEMENT NO.       4      6      5      7      8


   Messages:

   'ERROR' condition was raised
   Traceback of user routines:
   Compile Unit   Entry Statement   CU offset   Entry offset   Address
   Q              P2                +000001A0   +00000044      00020220
   Q              Q                 +000000CC   +000000C8      0002014C
```

Figure 4. Finding statement number from an entry offset in an error message

External symbol dictionary

If the option ESD applies, the compiler lists the contents of the external symbol dictionary (ESD).

The ESD is a table containing all the external symbols that appear in the object module. (The machine instructions in the object module are grouped together in *control sections*; an external symbol is a name that can be referred to in a control section other than the one in which it is defined.) The contents of an ESD appear under the following headings:

SYMBOL An 8-character field that identifies the external symbol.

TYPE Two characters from the following list to identify the type of entry:

- SD Section definition: the name of a control section within the object module.
- CM Common area: a type of control section that contains no data or executable instructions.

ER External reference: an external symbol that is not defined in the object module.

WX Weak external reference: an external symbol that is not defined in this module and that is not to be resolved unless an ER entry is encountered for the same reference.

PR Pseudoregister: a field used to address files, controlled variables, and FETCHed procedures.

LD Label definition: the name of an entry point to the external procedure other than that used as the name of the program control section.

ID Four-digit hexadecimal number: all entries in the ESD, except LD-type entries, are numbered sequentially, beginning with 0001.

ADDRESS Hexadecimal representation of the address of the external symbol.

LENGTH The hexadecimal length in bytes of the control section (SD, CM and PR entries only).

ESD entries

The external symbol dictionary usually starts with the standard entries shown in Figure 5, which assumes the existence of an external procedure called NAME.

SYMBOL	TYPE	ID	ADDRESS	LENGTH
CEESTART	SD	0001	000000	000080
***NAME1	SD	0002	000000	0000A8
***NAME2	SD	0003	000000	00005C
CEEMAIN	WX	0004	000000	
CEEMAIN	SD	0005	000000	000010
IBMRINP1	ER	0006	000000	
CEEFMAIN	WX	0007	000000	
CEEBETBL	ER	0008	000000	
CEEROOTA	ER	0009	000000	
CEESG010	ER	000A	000000	
NAME	LD		000008	

Figure 5. External symbol dictionary

*****name1**
SD-type entry for the program control section (the control section that contains the executable instructions of the object module). This name is the first label of the external procedure, padded on the left with asterisks to 7 characters if necessary, and extended on the right with the character 1.

*****name2**
SD-type entry for the static internal control section (which contains main storage for all variables declared STATIC INTERNAL). This name is the first label of the external procedure, padded on the left with asterisks to 7 characters if necessary, and extended on the right with the character 2.

CEESTART
SD-type entry for CEESTART. This control section transfers control to CEEROOTA, the initialization routine for the library environment. When initialization is complete, control passes to the address stored in the control section CEEMAIN. (Initialization is required only once while a PL/I program is running, even if it calls another external procedure. In such a case, control

passes directly to the entry point named in the CALL statement, and not to the address contained in CEEMAIN.)

CEEROOTA, CEESG010, CEEBETBL, IBMRINP1
These ER-type entries are generated to support environment initialization for the program.

The other entries in the external symbol dictionary vary, but can include the following:

- SD-type entry for the control section CEEMAIN, which contains the address of the primary entry point to the external procedure. This control section is present only if the procedure statement includes the option MAIN. A WX-type entry for CEEMAIN is always generated to support environment initialization for the program.
- Reference to a number of control sections as follows:

  CEEFMAIN — A control section used in *fetch* processing. It indicates the presence of a fetchable entry point within the load module. This will be the first OPTIONS(FETCHABLE) procedure in the phase—other procedures in the phase are not fetchable. A single phase may have a non-zero CEEFMAIN and CEEMAIN, and be both fetchable and invokable as a main phase.

  CEEUOPT — A control section that contains the run-time options specified at compile time.

  PLIXOPT — Run-time options string control section.
- LD-type entries for all names of entry points to the external procedure.
- ER-type entries for all the library subroutines and external procedures called by the source program.
- CM-type entries for variables declared STATIC EXTERNAL without the INITIAL attribute.
- SD-type entries for all other STATIC EXTERNAL variables and for external file names.
- PR-type entries for all file names. For external file names, the name of the pseudoregister is the same as the file name; for internal file names, the compiler generates pseudoregister names.
- PR-type entries for all controlled variables. For external variables, the name of the variable is used for the pseudoregister name; for internal variables, the compiler generates names.
- PR-type entries for fetched entry names.

Static internal storage map

The MAP option produces a Variable Offset Map. This map shows how PL/I data items are mapped in main storage. It names each PL/I identifier, its level, its offset from the start of the storage area in both decimal and hexadecimal form, its storage class, and the name of the PL/I block in which it is declared.

If the LINECOUNT option specifies a value less than 100, the static internal storage map is printed in two columns across the page. If LINECOUNT is 100 or greater,

the map will be printed as a single column. This makes it more suitable for viewing at a terminal.

If the LIST option is also specified a map of the static internal and external control sections is also produced.

For more information about the static internal storage map and an example, see the *LE/VSE Debugging Guide and Run-Time Messages.*

Object listing

If the option LIST applies, the compiler generates a listing of the machine instructions of the object module, including any compiler-generated subroutines, in a form similar to the Assembler language.

If the LINECOUNT option specifies a value less than 100, the object listing is printed in two columns across the page. If LINECOUNT is 100 or greater, the list will be printed as a single column. This makes it more suitable for viewing at a terminal.

For more information about the object listing and an example, see the *LE/VSE Debugging Guide and Run-Time Messages.*

Messages

If the preprocessor or the compiler detects an error, or the possibility of an error, they generate messages. 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. All messages are graded according to their severity, as follows:

I An information message that calls attention to a possible inefficiency in the program or gives other information generated by the compiler.

W A warning message that calls attention to a possible error, although the statement to which it refers is syntactically valid.

E An error message that describes an error detected by the compiler for which the compiler applied a *fix-up* with confidence. The resulting program will run, and it will probably give correct results.

S A severe error message that specifies an error detected by the compiler for which the compiler cannot apply a *fix-up* with confidence. The resulting program will run but will not give correct results.

U An unrecoverable error message that describes an error that forces termination of the compilation.

The compiler only lists messages that have a severity equal to or greater than that specified by the FLAG option, as shown in Table 6 on page 43.

Each message is identified by an eight-character code of the form *IELnnnnI*, where:

- The first three characters *IEL* identify the message as coming from the compiler.
- The next four characters, *nnnn*, are a four-digit message number.
- The last character, *I*, is an operating system code for the operator indicating that the message is for information only.

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

Table 6. Using the FLAG option to select the lowest message severity listed

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

Return codes

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. 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. The meanings of the codes are given in Table 7.

Table 7. Return codes from compilation of a PL/I program

Return code	Description
0000	No error detected, but informational messages may be produced; compilation completed, successful execution anticipated.
0004	Warning; possible error detected; compilation completed, execution probable.
0008	Error detected; compilation completed; successful execution probable.
0012	Severe error detected; compilation not necessarily completed; successful execution improbable.
0016	Unrecoverable error detected; compilation terminated abnormally; successful execution impossible.

Chapter 2. Compiling under VSE

This chapter describes the job control statements used for compiling PL/I programs under VSE.

You should read this chapter in conjunction with the *VSE/ESA System Control Statements* book, which gives a full description of all the VSE JCL statements. It also describes the operation of the linkage editor and the VSE Librarian.

Job control for compilation

You compile a PL/I program by using VSE JCL statements. The statement that initiates the PL/I compiler is

```
// EXEC IEL1AA,SIZE=nK
```

If you want to save the object code that the compiler generates, you should place an OPTION statement before the EXEC statement, as follows:

```
// OPTION DECK
// EXEC IEL1AA,SIZE=nK
```

The compiler will write the object code onto the data set identified by SYSPCH in your JCL. If you want this object code to be kept in a library, you can use the compiler NAME option (see NAME on page 16). This tells the compiler to place a VSE Librarian CATALOG command card as the first record on the SYSPCH file. You can then follow your compile JCL with some Librarian JCL to save the object code.

If you want to follow the compilation job step with a link-editing job step, you should specify the LINK option, as follows:

```
// OPTION LINK
// EXEC IEL1AA,SIZE=nK
```

In addition, if you want the link-edited program (or phase) to be permanently stored in a library, you should specify the CATAL option, as follows:

```
// OPTION CATAL
// EXEC IEL1AA,SIZE=nK
```

The CATAL option also sets the LINK option, so there is no need to specify both. The CATAL and LINK options tell the compiler to write the object code onto the data set identified by SYSLNK. The linkage editor can then read this data set to create an executable phase.

If you use OPTION CATAL, you can also use the compiler NAME option to identify the phase name of the executable program (see NAME on page 16). This tells the compiler to write a linkage editor PHASE statement as the first record on the SYSLNK file. The linkage editor then uses this statement to name the executable module in the library.

Compiler data sets

The compiler uses standard device assignments for its data sets, but you can modify those assignments if there are special requirements for compiler input/output. For example, if the source module is to be read from magnetic tape or if the object module written on SYSPCH is required on magnetic tape, the symbolic device name can be assigned accordingly by means of the JCL ASSGN statement.

The compiler requires several standard data sets. These are shown in Table 8, and described in the following paragraphs.

Table 8. Compiler standard data sets

Symbolic name	Function	Device type	File	When required
SYSIPT	Input to the compiler	DASD Magnetic tape Card reader Diskette	IJSYSIN	Always
SYSLNK	Object module	DASD	IJSYSLN	When link-editing follows compilation in the same job
SYSPCH	Preprocessor output, Compiler output	DASD Magnetic tape Card punch Diskette	IJSYSPH	Preprocessor: when program source needs to be saved Compiler: when link-editing takes place in a subsequent job
SYS001	Temporary workfile	DASD	IJSYS01	When the partition is too small to complete the compile in memory
SYSLST	Listing, including messages	DASD Magnetic tape Printer Diskette	IJSYSLS	Always
(LIBDEF defines the library name)	Included source statements	DASD		When %INCLUDE is used

Primary input (SYSIPT)

The primary input to the compiler must be a consecutive data set containing your PL/I source program. The first statement in the file can be a compiler-control PROCESS statement (either %PROCESS or *PROCESS). This statement is used to specify the compiler options to be used for the compilation. The source module can comprise one or more external procedures. If you want to compile more than one external procedure in a single job step, separate the external procedures in the input data set with PROCESS statements. (This use of the PROCESS statement is described under "Compiling multiple procedures in a single job step" on page 48.)

The input data set may be on a diskette, direct-access device, magnetic tape, or punched cards. The address of the device used must be assigned to SYSIPT.

Output (SYSLNK or SYSPCH)

The compiler can optionally transmit the object module to a data set on SYSLNK, a data set on SYSPCH, or both. The object module is always in the form of 80-byte fixed-length records, blocked or unblocked. This form is suitable for processing by the VSE linkage editor program.

If you specify the JCL option LINK or CATAL, the compiler will transmit the object module to a data set on SYSLNK. If you specify DECK, the object module will go to SYSPCH.

If you specify the MDECK compile-time option, the compiler will also store the output from the preprocessor on the SYSPCH data set.

Listing (SYSLST)

The compiler generates a listing that can include 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 appropriate compile-time options, are described under "Using the compiler listing" on page 32.

The symbolic file used for the listing is SYSLST. The first character of each record is an American National Standard carriage control character. The records are 133 bytes, fixed length, but if the SYSLST data set is on a CKD or ECKD* DASD device, they will be truncated to 121 bytes.

Temporary workfile (SYS001)

The compiler normally requires a data set for use as a temporary workfile (it will not be needed if the compile is small enough to fit into the storage in the partition). This data set is known as the *spill file*, and it must be on a direct access device, which is assigned to SYS001.

The spill file is used as a logical extension to main storage and is used by the compiler and the preprocessor to contain text and dictionary information.

Statement lengths: The compiler has a restriction that any statement must fit into the compiler's work area. This restricts the maximum length of a PL/I statement to 3400 characters.

The DECLARE statement is an exception in that it can be regarded as a sequence of separate statements, each of which starts wherever a comma occurs that is not contained within parentheses. For example:

```
DCL 1 A,
      2 B(10,10)  INIT(1,2,3,...),
      2 C(10,100) INIT((1000)(0)),
      (D,E) CHAR(20) VAR,...
```

In this example, each line can be treated by the compiler as a separate DECLARE statement in order to accommodate it in the work area. The compiler will also treat the INITIAL attribute in the same way when it is followed by a list of items separated by commas that are not contained within parentheses. Each item can contain initial values that, when expanded, do not exceed the maximum length. The above also applies to the use of the INITIAL attribute in a DEFAULT statement.

If a DECLARE statement cannot be compiled, the following techniques are suggested to overcome this problem:

- Simplify the DECLARE statement so that the compiler can treat the statement in the manner described above.
- Modify any lists of items following the INITIAL attribute so that individual items are smaller and separated by commas not contained in parentheses. For example, the following declaration is followed by an expanded form of the same declaration. The compiler can more readily accommodate the second declaration in its work area:

```
1.  DCL Y (1000) CHAR(8)
    INIT ((1000) (8)'Y');

2.  DCL Y (1000) CHAR(8) INIT
    ((250)(8)'Y',(250)(8)'Y',
     (250)(8)'Y',(250)(8)'Y');
```

Source statement library

If you use the %INCLUDE statement to introduce source statements into the PL/I program from a library, you must define the library using a LIBDEF JCL statement. The compiler will then search the specified library chain for the module named in the %INCLUDE statement.

Specifying compiler 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 the JCL 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 (except that CONTROL, if used, must be first), separating the options with commas and enclosing the list within single quotation marks. For example:

```
// EXEC IEL1AA,SIZE=128K,PARM='OPTIONS,LIST,XREF(FULL)'
```

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 extend the first line up to column 71, place a non-blank continuation character (usually a C) in column 72, and continue on the next line from column 16. For example:

```
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
// EXEC IEL1AA,SIZE=128K,PARM='AGGREGATE,ATTRIBUTES(SHORT),C,ESD,F(I),MC
               ACRO,MI(''X''),NEST,STG,X'
```

Compiling multiple procedures in a single job step

Batched compilation allows the compiler to compile more than one external PL/I procedure in a single job step. The compiler creates an object module for each external procedure and stores it sequentially either in the SYSLNK or SYSPCH data set. Batched compilation can increase compiler throughput by reducing operating system and compiler initialization overheads.

To specify batched compilation, include a compiler %PROCESS statement as the first statement of each external procedure except possibly the first. The %PROCESS statements identify the start of each external procedure and allow compile-time options to be specified individually for each compilation. The first procedure might require a %PROCESS statement of its own, because the options in the PARM parameter of the EXEC statement apply to all procedures in the batch, and can conflict with the requirements of subsequent procedures.

Note: The options specified in the %PROCESS statement override those specified in the PARM parameter of the EXEC statement.

The method of coding a %PROCESS statement and the options that can be included are described under "Specifying options in the %PROCESS or *PROCESS statements" on page 27. The options specified in a %PROCESS statement apply to the compilation of the source statements between that %PROCESS statement and the next %PROCESS statement. Options other than these, either the defaults or those specified in the PARM field, will also apply to the compilation of these source statements. Two options, the SIZE option and the NAME option have a particular significance in batched compilations, and are discussed below.

Note: The MDECK option can cause problems if it is specified on second or subsequent compilations but not on the first. This is because it requires the opening of SYSPCH and there might not be room for the associated data management routines and control blocks. When this happens, compilation ends with a storage abend. To overcome this problem, subtract 4K from the SIZE option storage specification, and rerun the compile job.

SIZE option

In a batched compilation, the SIZE specified in the first procedure of a batch (by a %PROCESS or EXEC statement, or by default) is used throughout. If SIZE is specified in subsequent procedures of the batch, the compiler prints a diagnostic message and ignores the specification.

NAME option

The NAME option, when used with the JCL OPTION CATAL statement, specifies that the compiler places a linkage editor PHASE statement as the first statement of the object module. The use of this option in the PARM parameter of the EXEC statement, or in a %PROCESS statement, determines how the object modules produced by a batched compilation are handled by the linkage editor. When the batch of object modules is link-edited, the linkage editor combines all the object modules from one PHASE statement to the next into a single load module, and takes the name of the load module from the first PHASE statement. When combining two object modules into one load module, if the NAME option is used in the EXEC statement, the result is the same as if it were specified in the %PROCESS statement for the first source program. An example of the use of the NAME option is given in Figure 6.

```
// OPTION CATAL
// EXEC   IEL1AA,SIZE=128K,PARM='LIST,NAME(''A'')'
   ALPHA: PROC OPTIONS(MAIN);
             .
             .
             .
          END ALPHA;
%  PROCESS;
   BETA: PROC;
             .
             .
             .
         END BETA;
%  PROCESS NAME('C');
   GAMMA: PROC;
             .
             .
             .
          END GAMMA;
/*
```

Figure 6. Use of the NAME option in batched compilation

Compilation of the PL/I procedures ALPHA, BETA, and GAMMA results in the following object modules and PHASE statements:

```
          PHASE A,*
   OBJECT MODULE FOR ALPHA
   OBJECT MODULE FOR BETA
          PHASE C,*
   OBJECT MODULE FOR GAMMA
```

From this sequence of object modules and control statements, the linkage editor produces two executable phases, one named A containing the object modules for the external PL/I procedures ALPHA and BETA, and the other named C containing the object module for the external PL/I procedure GAMMA.

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. See "Return codes" on page 43 for a description of the return codes that the compiler generates.

Examples of batched compilations

Figure 7 is an example of simple batched compilation. It illustrates the use of a single invocation of the PL/I compiler to compile three procedures.

```
// JOB    OPT4#14
// OPTION CATAL
// EXEC   IEL1AA,SIZE=128K,PARM='NAME(''PROG1,*'')'
          First PL/I source program
% PROCESS;
          Second PL/I source program
% PROCESS;
          Third PL/I source program
/*
// EXEC   LNKEDT
/&
```

Figure 7. Example of batched compilation

The three external procedures will be link-edited together to form a load module called PROG1.

Correcting compiler-detected errors

At compile time, both the preprocessor and the compiler can produce diagnostic messages and listings according to the compile-time options selected for a particular compilation. The listings and the associated compile-time options are discussed in Chapter 1, "Using compile-time options and facilities" on page 4. The diagnostic messages produced by the compiler are identified by a number with an "IEL" prefix. Each message is listed in *PL/I VSE Compile-Time Messages and Codes.* This publication includes explanatory notes, examples, and any action to be taken.

You should always check the compilation listing for occurrences of these messages to determine whether the syntax of the program is correct. Messages of greater severity than warning (that is, error, severe error, and unrecoverable error) should be acted upon if the message does not indicate that the compiler has been able to "fix" the error correctly. You should be aware that the compiler, in making an assumption as to the intended meaning of any erroneous statement in the source program, can introduce another, perhaps more severe, error which in turn can produce yet another error, and so on. When this occurs, the compiler produces a number of diagnostic messages which are all caused either directly or indirectly by the one source error.

Other useful diagnostic aids produced by the compiler are the attribute table and cross-reference table. The attribute table, specified by the ATTRIBUTES option, is useful for checking that program identifiers, especially those whose attributes are contextually and implicitly declared, have the correct attributes. The cross-reference table is requested by the XREF option, and indicates, for each program variable, the number of each statement that refers to the variable.

To prevent unnecessary waste of time and resources during the early stages of developing programs, use the NOOPTIMIZE, NOSYNTAX, and NOCOMPILE options. The NOOPTIMIZE option suppresses optimization unconditionally, and the remaining options suppress compilation, link-editing, and execution if the appropriate error conditions are detected.

CICS considerations

PL/I language restrictions

When designing a PL/I application to run under CICS, you must be aware of the restrictions on some of the PL/I language constructs for the CICS environment. The *LE/VSE Programming Guide* contains a list of PL/I facilities that are not supported under CICS.

Compiling for CICS

Prior to compiling your transaction, you must invoke the CICS Command Language Translator. You can find information on the CICS Command Language Translator in the *CICS/VSE Application Programmer's Reference Manual.* After the CICS translator step ends, compile your PL/I program with the SYSTEM(CICS) option. For a description of the SYSTEM compile-time option, see "SYSTEM" on page 24.

Run-time environment

The *LE/VSE Programming Guide* also lists some considerations for running PL/I applications under CICS, including run-time options, storage management, and condition handling.

Chapter 3. The VSE Librarian

The VSE Librarian is a component of the VSE/ESA operating system that maintains programs in libraries. You can use the Librarian to keep your PL/I INCLUDE members, and PL/I can use the Librarian to store compiler output (object code). The linkage editor uses the Librarian to store executable programs (phases), and you can then use the Librarian to maintain those phases (delete, rename, etc).

You can find a full description of the Librarian in the *VSE/ESA System Control Statements* book. This chapter will give you a brief overview, and will show you how to use some of the Librarian functions with your PL/I programs.

Library members

The items contained within libraries are called *members*, and each member is identified by its *member name* and *member type*. Member names can be up to eight characters long, and are usually equated to a program name. Member types can also be up to eight characters long, and the system recognizes certain predefined types or categories of members. The member types that you will need to know about are:

P Member types of a single character are reserved for program source. This is used as input to a language compiler such as PL/I. Member type P signifies PL/I source.

OBJ This is used for object code produced by the compiler. The linkage editor reads OBJ members and combines them to create an executable program phase.

PHASE This identifies an executable program created by the linkage editor. This is ready for loading into storage and running.

The full member name consists of both the name and the type joined together by a period, for example:

PROGA.P	is the name of a PL/I source member,
PROGA.OBJ	is the member containing the compiled object code, and
PROGA.PHASE	is the executable code.

Libraries and sublibraries

Your system can contain any number of *libraries*, and each library can contain any number of *sublibraries*. The members are contained within the sublibraries. When you use the Librarian, you specify the library and sublibrary name that you are working with, and then you can perform Librarian operations on the members in that sublibrary.

Library names are up to seven characters long, and sublibrary names are up to eight characters. You identify the sublibrary that you are working with by specifying the library name and sublibrary name connected by a period, for example USERLIB.ORDERS.

The *VSE/ESA System Control Statements* book describes how to define new libraries and sublibraries, but your installation should have predefined libraries for you to use.

Using the Librarian

You invoke the Librarian program with a JCL EXEC statement, specifying LIBR as the program name, as:

```
// EXEC LIBR
```

You can use the PARM option of the EXEC statement to specify commands for the Librarian to perform. For example:

```
// EXEC LIBR,PARM='ACCESS SUBLIB=USERLIB.SALES'
```

You can also code Librarian commands in the SYSIPT file. This would normally follow the EXEC statement in the job input stream, but you could assign SYSIPT to another device or data set, and the Librarian will read its commands from there. The Librarian will perform the commands on the PARM option first, and then it will do the commands from the SYSIPT file.

Librarian commands

The *VSE/ESA System Control Statements* book contains a full description of the Librarian and all of its commands. The main commands that you will be concerned with are ACCESS, CATALOG, DELETE, LIST, and LISTDIR. These are discussed here.

Note: The syntax given in this chapter is not complete. The Librarian commands shown here have other options and parameters that are described in the *VSE/ESA System Control Statements* book.

ACCESS

This command identifies the library and sublibrary that you want to work with. It is normally the first Librarian command that is processed, because all the following commands affect the sublibrary that is specified on the ACCESS command.

The syntax of the command is:

```
ACCESS SUBLIB=library.sublib
```

The library name can be up to seven characters long, and the sublibrary name can be up to eight characters. These will normally be predefined at your installation.

CATALOG

The CATALOG command is used the create a new member or replace an existing member in a sublibrary. Its syntax is:

```
CATALOG member.type [EOD=xx] [REPLACE={NO|YES}]
```

member
The name of the member to be entered or replaced in the library. This can be from one to eight characters long.

type
The member type. This can also be up to eight characters long, but will usually be one of the types listed under "Library members" above.

EOD=xx
Two characters that will identify the end of the input data. When the Librarian encounters these characters in columns one and two of an input record, it terminates processing for this member. The default is a slash and a plus sign (/+) for most member types, but you can change this if your input file contains /+ as part of its data.

REPLACE=NO|YES
The member replacement option. If the member already exists, and REPLACE=YES is specified, the existing member will be deleted and replaced with the input data.

DELETE

You can delete members from libraries using the DELETE command. Its syntax is:

```
DELETE member.type
```

LIST

You can use the LIST command to list the contents of a member. Its syntax is:

```
LIST member.type
```

LISTDIR

The LISTDIR command lists the contents of library directories. The output from LISTDIR is a list of member names sorted in alphanumeric collating sequence. The syntax of the LISTDIR command is:

```
LISTDIR {LIB=library | SUBLIB=library.sublibrary | member.type }
```

LIB=library
List the directory contents of a library. This includes all of its sublibraries.

SUBLIB=library.sublibrary
List the directory contents of the specified sublibrary. For this listing, the primary sort sequence is the member type, so that the members are grouped by type.

name.type
List the directory information for the specified member. With this form of the command, you can use an asterisk (*) as a generic name specification, so you can produce a list of matching member names. For example:

```
LISTDIR *.P          will list all members of type P,
LISTDIR PROGA.*      will list all members with a name of PROGA,
LISTDIR AB*.OBJ      will list all members with a type of OBJ and
                     a name that begins with AB.
```

Cataloging source members

You can use the Librarian to keep PL/I source members, which you can then include in your program with a %INCLUDE statement. The following example shows a job that creates a library member called DCLIB.

```
// JOB  SOURCE
// EXEC LIBR
ACCESS SUBLIB=USERLIB.FILES
CATALOG DCLIB.P REPLACE=YES
 /* Standard declaration of DCLIB */
 DCL 1 RECQ,
       2 (RECA,
          RECB,
          RECC,
          RECD) CHAR(50),
       2 KEY,
         3 (FIRST,
            SECOND,
            THIRD) CHAR(3),
         3 CODE CHAR(1);
 /* End of DCLIB */
/+
/*
/&
```

To include this member in a PL/I program, you need to identify the library and sublibrary name in your compile JCL, and code a %INCLUDE statement in your program. For example:

```
// JOB    COMPILE
// LIBDEF *,SEARCH=(USERLIB.FILES)
// EXEC   IEL1AA,SIZE=128K,PARM='INCLUDE,SOURCE,XREF'
 PROGA: PROC OPTIONS (MAIN);
        .
        .
        .
 %INCLUDE P(DCLIB);
        .
        .
        .
 END PROGA;
/*
/&
```

In this example:

- The LIBDEF JCL statement identifies the name of the library and sublibrary where the INCLUDE member can be found;
- The PL/I compiler option INCLUDE is specified (in the PARM option of the EXEC IEL1AA statement). This allows the source program to use %INCLUDE statements; and
- The %INCLUDE P(DCLIB) statement in the source program specifies the member to be included (DCLIB.P).

Cataloging object code

You can use the Librarian to keep object code produced by the PL/I compiler. The linkage editor can then read the library to find its required object modules, and combine them to form an executable phase.

You use the JCL OPTION statement (OPTION DECK) to tell the compiler to produce object code output, and the compiler NAME option to name the object member. For example:

```
// JOB    COMPOBJ
// OPTION DECK
// EXEC   IEL1AA,SIZE=128K
 %PROCESS NAME('NEWPROG');
 NEWPROG: PROC OPTIONS(MAIN);
      .
      .
      .
 END;
/*
/&
```

In this example:

- The JCL OPTION DECK statement tells the compiler to produce object code output. This will go to the SYSPCH file.
- The compiler NAME option (specified on the %PROCESS statement) tells the compiler the name of the member to contain the object module. The compiler will write a Librarian CATALOG command as the first record on the SYSPCH file, before the object code itself. The member name on the CATALOG command will be NEWPROG (from the NAME option), and the member type will be OBJ. The format of the command is:

  ```
  CATALOG NEWPROG.OBJ,REPLACE=YES
  ```

Note: For a member type of OBJ, the Librarian does not require an end-of-input (/+) statement, so the compiler does not produce one, unless this is a batched compilation, in which case an end-of-input (/+) statement is generated before the second or subsequent CATALOG statements. An end-of-input (/+) is not generated for the last object output.

The compiler does not write its object code directly into a library—it writes it onto the file identified by SYSPCH in your JCL. You should follow the compile step in your JCL with a Librarian step, specifying SYSIPT as the same data set (the SYSPCH compiler output). This will catalog the object code into the library with the specified member name.

Figure 8 on page 57 contains an example of compiling a program and cataloging its object code into a library. The program is a PL/I 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. By saving the object code for this program in a library, you make it available for other programs to use without having to recompile it.

```
// JOB    COMPOBJ
// OPTION DECK
// DLBL   IJSYSPH,'PROGRAM.OBJECT',0,SD
// EXTENT SYSPCH,VSE111,1,0,1200,20
ASSGN     SYSPCH,DISK,VOL=VSE111,SHR
// EXEC   IEL1AA,SIZE=128K
%PROCESS NAME('ELAPSE');
 ELAPSE:  PROC(TIME1,TIME2) RETURNS(DEC FIXED(7));
   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;
/*
CLOSE     SYSPCH,PUNCH
// DLBL   IJSYSIN,'PROGRAM.OBJECT',0
// EXTENT SYSIPT
ASSGN     SYSIPT,DISK,VOL=VSE111,SHR
// EXEC   LIBR,PARM='ACCESS SUBLIB=USERLIB.UTILS'
/*
CLOSE     SYSIPT,SYSRDR
/&
```

Figure 8. Compiling a program and cataloging the object code

In this example:

- The OPTION DECK JCL statement tells the compiler to write an object code file.
- The NAME compiler option tells the compiler to write a Librarian CATALOG command as the first record on the file. The command format will be:

  ```
  CATALOG ELAPSE.OBJ REPLACE=YES
  ```

- The DLBL, EXTENT, and ASSGN statements identify the name of the VSE data set (PROGRAM.OBJECT) that will be used as the SYSPCH file.
- After the compile, the DLBL IJSYSIN statement (and the EXTENT and ASSGN statements) identify the same VSE data set (PROGRAM.OBJECT) as the input file for the Librarian.
- The Librarian first executes the ACCESS command (from the PARM option of the EXEC LIBR statement), which defines the library and sublibrary to be used.
- The Librarian then executes the CATALOG command from its SYSIN file, and creates a member called ELAPSE.OBJ in the sublibrary USERLIB.UTILS.

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 modules in the LE/VSE run-time library. These references are resolved during link-editing or during execution (dynamically).

After you compile your PL/I program, the next step is to link-edit the program, and then run it with test data to verify that it produces the results you expect.

LE/VSE provides the run-time environment and services you need to execute your program. For instructions on linking and running PL/I and all other LE/VSE-conforming language programs, refer to the *LE/VSE Programming Guide.* For information about migrating your existing PL/I programs to LE/VSE, see the *PL/I VSE Migration Guide.*

This chapter contains the following sections:

The VSE linkage editor

The linkage editor has two major functions. One is to resolve any unresolved addresses in the PL/I object module, to enable it to be loaded and executed in the required main storage partition; and the other is to incorporate any required object modules from the library search sequence (such as PL/I or LE/VSE library routines).

The resultant output from the linkage editor is an executable program phase. This output is written into a library, from where it can be subsequently loaded for execution.

Input to the linkage editor

The linkage editor can accept input as follows:

- Linkage editor control statements from SYSIPT, SYSRDR, or SYSLNK. Control statements can also be included in object modules loaded from a library.
- Relocatable object modules from:
  - SYSLNK (compiler output)
  - any of the libraries specified in the LIBDEF search sequence

Output from the linkage editor

The linkage editor produces the following outputs:

- An executable program phase, either catalogued permanently in a library or stored temporarily for immediate execution,
- A module map, and if necessary, diagnostic messages to indicate any detected error conditions, on SYSLST.

Linkage editor processing

The *LE/VSE Programming Guide* describes the operation of the linkage editor and how it relates to the LE/VSE environment. It also describes how to link routines written in PL/I with routines written in other LE/VSE-conforming languages.

The *VSE/ESA System Control Statements* book contains a full description of the JCL and control statements required to run the VSE linkage editor.

Selecting math results at link-edit time

The mathematical routines used by PL/I VSE are supplied as part of the LE/VSE product, and are also available to other LE/VSE-conforming languages. When your program uses PL/I mathematical built-in functions, PL/I uses the LE/VSE routines to perform the mathematical operations. These routines are similar to those supplied with the DOS PL/I Optimizing Compiler. The range and precision of these functions and the accuracy of the results will generally be the same as that of the DOS PL/I routines. However, there might be slight differences in the results produced by some of the functions.

Because the LE/VSE routines are defaults, if you recompile a DOS PL/I application with PL/I VSE, the program will be link-edited with the LE/VSE routine stubs. If you wish to ensure that the results of mathematical operations are consistent with DOS PL/I, you can select DOS PL/I-compatible routines at link-edit time to override the default LE/VSE routines.

To select DOS PL/I-compatible results, you need to ensure that the stubs for the PL/I math routines are linked into the application. You can do this with an INCLUDE statement in your link-edit JCL, as follows:

```
INCLUDE IBMSDOSM
```

This will ensure that the DOS PL/I-compatible routine stubs are link-edited with your program instead of the standard LE/VSE routine stubs.

Notes:

1. The DOS PL/I-conforming routines are supplied for compatibility only.
2. The *LE/VSE Programming Guide* contains a description of each mathematical routine, along with the range of results for numbers of different precision.
3. The choice of routines affects PL/I built-in functions, and PL/I language elements such as exponentiation.
4. The choice of routines applies to all PL/I procedures that are linked into a phase—different subroutines in the same phase must use the same mathematical routines. However, a phase linked with one set of routines can FETCH a phase linked with the other set.

Program run-time considerations

After you compile and link-edit your program, it is ready to be run in a VSE partition. You use VSE job control (JCL) statements to initiate the execution of your program. The *VSE/ESA System Control Statements* book contains a full description of all the JCL statements.

Specifying run-time options

In the LE/VSE environment, you can specify run-time options that will control the execution of your program. You can specify options such as where storage will be allocated from for automatic variables, and whether or not a storage usage report will be produced when the program has finished.

Note: The *LE/VSE Programming Guide* contains a full description of all the options available, and a list of the places where you can specify them. We will only look at specifying them in your JCL here.

You can specify run-time options in the PARM string of the EXEC statement in your JCL, as follows:

```
// EXEC MYPROG,PARM='ALL31(ON),RPTSTG(ON)/'
```

The slash at the end of the PARM string is used to separate the run-time options from the program parameter, which is discussed next.

The LE/VSE run-time system interprets the PARM string, up to the slash, and uses the options specified to set up the environment for running your program. The *LE/VSE Programming Guide* describes all of the available options, and the effect that they have on running your program.

Passing parameters to your program

You can also use the PARM string in your JCL to pass a parameter to your program. The parameter will be given to your PL/I MAIN procedure as an argument. Its format will be a varying-length character string, with a maximum length of 100 bytes.

You use the PARM string of the EXEC JCL statement to specify either the run-time options or the program parameter, or both. The slash is used to separate the two. For example, to specify *both* run-time options *and* a program parameter, you could code your EXEC statement as follows:

```
// EXEC MYPROG,PARM='ALL31(ON),RPTSTG(ON)/TUESDAY'
```

LE/VSE will use the part up to the slash as run-time options, and will give the part after the slash to your program as a varying-length character string with a current length of 7 and a value of 'TUESDAY'.

If you don't want to specify any run-time options, but you do want to pass a parameter to your program, you can code your JCL as follows:

```
// EXEC MYPROG,PARM='/TUESDAY'
```

or

```
// EXEC MYPROG,PARM='TUESDAY'
```

You need to take the following actions to allow your program to receive the parameter:

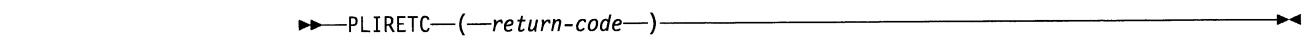
- Code your program with a varying-length character string as a parameter to the MAIN procedure. The maximum length can be between 1 and 100. For example:

```
MYPROG: PROCEDURE(DAY_OF_WEEK) OPTIONS(MAIN);
                .
DCL DAY_OF_WEEK      CHAR(10)  VARYING;
                .
                .
```

- Compile your program with the SYSTEM(VSE) compiler option. This ensures that the format of the argument constructed from the JCL parameter matches that of the program variable.
- Code the PARM option in your JCL with the data to be passed to your program.

Setting the system return code

Your PL/I program can set a value in the system return code, which can then be checked in VSE JCL to provide a conditional execution facility. You set the return code by calling the PLIRETC built-in subroutine, which has the following syntax:



return-code
is a constant or variable with attributes FIXED BIN(31) that contains the value of the return code to be set

When PLIRETC is called from the main procedure, PL/I will set the system return code to the value in the *return-code* parameter. For example:

```
PROG1: PROCEDURE OPTIONS(MAIN);
DCL PLIRETC BUILTIN;
    .
    .
IF ANY_ERRORS THEN
   CALL PLIRETC(8);
ELSE
   CALL PLIRETC(0);
RETURN;
END PROG1;
```

The JCL can then check the return code to determine whether to execute the next job step. For example:

```
/. DOPROG1
// DLBL ...
// EXTENT ...
// EXEC PROG1
/*
/. DOPROG2
IF $RC NE 0 THEN
GOTO DOPROG3
// DLBL ...
// EXTENT ...
// EXEC PROG2
/*
/. DOPROG3
   .
   .
```

This JCL will only execute PROG2 if the return code from PROG1 is equal to zero.

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*. On your VSE/ESA system, a collection of records is called a *data set* or *file*. In PL/I terminology, we always use the term *data set* to refer to a collection of records. 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 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.

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: this is its *data set name*. In some cases such as print files and unlabeled tape files the data set has no name; it is known to the system only 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 provide a statement, external to your program, that associates the PL/I file name with a data set name. This is done in your VSE JCL by means of the job control statements ASSGN, DLBL, and TLBL. For example, if you have the following file declaration in your program:

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

you should create a DLBL statement with a *filename* that matches the name of the PL/I file, and a physical data set name (file-id) of the data set that contains the actual data records.

```
// DLBL STOCK,'PARTS.INSTOCK'
```

The common name that relates the DLBL statement with the PL/I file (STOCK in this example) is called a *file reference*. This is the item that must have the same name both inside and outside the program, so that it can provide the required relationship.

You'll find some guidance in writing JCL statements in this manual, but for more detail refer to the *VSE/ESA System Control Statements* book.

Associating data sets with files

You associate a data set with a PL/I file by ensuring that the file reference on the DLBL statement that defines the data set is the same as *either*:

- The first seven characters of the declared PL/I file name (padded or truncated), *or*
- 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 file references 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 file reference defaults to the PL/I file name. If the file name is longer than 7 characters, the file reference is composed of the first 7 characters of the file name.
- The character set of the job control language does not contain the break character (_). Consequently, this character cannot appear in the first 7 characters of a file name, unless the file is to be opened with a TITLE option with a valid name as its expression. The alphabetic extender characters $, @, and #, however, are valid for file references.

Since PL/I external names[1] 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 file reference in the associated JCL 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 file reference of a DLBL statement in the current job step. When statement number 2 is run, the name OLDMAST is taken to be the same as the file reference of another DLBL statement in the current job step. (The first 7 characters of a file name form the file reference. If OLDMASTER is a PL/I 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 file reference of a DLBL statement in the current job step.

In each of the above cases, a corresponding JCL statement must appear in the job stream; otherwise, the UNDEFINEDFILE condition is raised. These could be DLBL statements, as follows:

```
1. // DLBL MASTER,'MASTER.CURRENT' ...
2. // DLBL OLDMAST,'MASTER.OLD' ...
3. // DLBL DETAIL,'DETAIL.NEW' ...
```

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

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

---

1 PL/I external names are the names of PL/I entities (variables, files, procedures, etc) that are declared with the EXTERNAL attribute, and are therefore known by more than one PL/I procedure. File references are the external names used to relate PL/I files to VSE data sets. These are two different things.

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

```
// DLBL RPRICE,...
```

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 DLBL statement in the current job step with DETAIL1 as its filename. It could start as follows:

```
// DLBL DETAIL1,'DETAILA' ...
```

Thus, you associate the *data set* DETAILA with the *file* DETAIL through the *file reference* 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 physical data set at the same time. This is illustrated in the following example, where INVNTRY is the filename of a DLBL statement that defines 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 any data set damage that might occur.

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 DETAIL is associated with the data set named on the DLBL JCL statement with the filename DETAIL1. If you closed and reopened the file, you could specify a different name 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('MASTER'||IDENT);
    .
    .
    .
   CLOSE FILE(MASTER);
END;
```

In this example, when MASTER is opened during the first iteration of the do-group, the associated file reference is taken to be MASTERA. After processing, the file is closed, dissociating the file reference from the file. During the second iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the file reference MASTERB. Similarly, during the final iteration of the do-group, MASTER is associated with the file reference MASTERC.

Your JCL should contain three DLBL statements, one for each data set. Even though the file references in the three DLBL statements will be different (MASTERA, MASTERB, and MASTERC), the TITLE option of the OPEN statement allows the same PL/I file to be used for all three data sets.

Establishing data set characteristics

A data set consists of records stored in a particular format which the operating system understands. 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.

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 specify the block size in the BLKSIZE option of the ENVIRONMENT attribute.

A *record* is the unit of data transmitted to and from a program. You specify the record length 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 operating system's data management routines.

Information interchange codes: The normal code in which data is recorded is the Extended Binary Coded Decimal Interchange Code (EBCDIC). However, for magnetic tape only, the operating system accepts data recorded in the American Standard Code for Information Interchange (ASCII). You use the ASCII and BUFOFF options of the ENVIRONMENT attribute if your program will read or write data sets recorded in ASCII.

A prefix field up to 99 bytes in length might be present at the beginning of each block in an ASCII data set. The use of this field is controlled by the BUFOFF option of the ENVIRONMENT attribute. For a full description of the ASCII option, see "ASCII" on page 101.

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, if required.

You specify the record format 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 9 on page 71, 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.

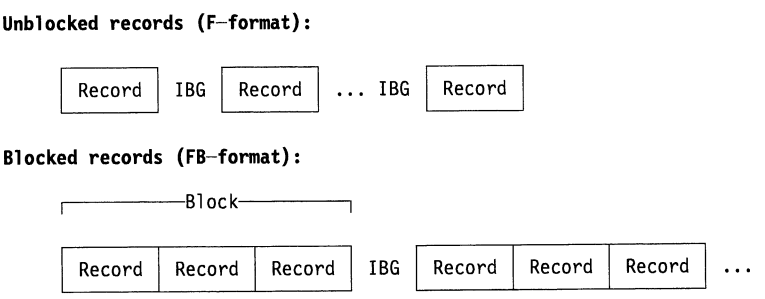


Figure 9. Fixed-length records

Variable-length records

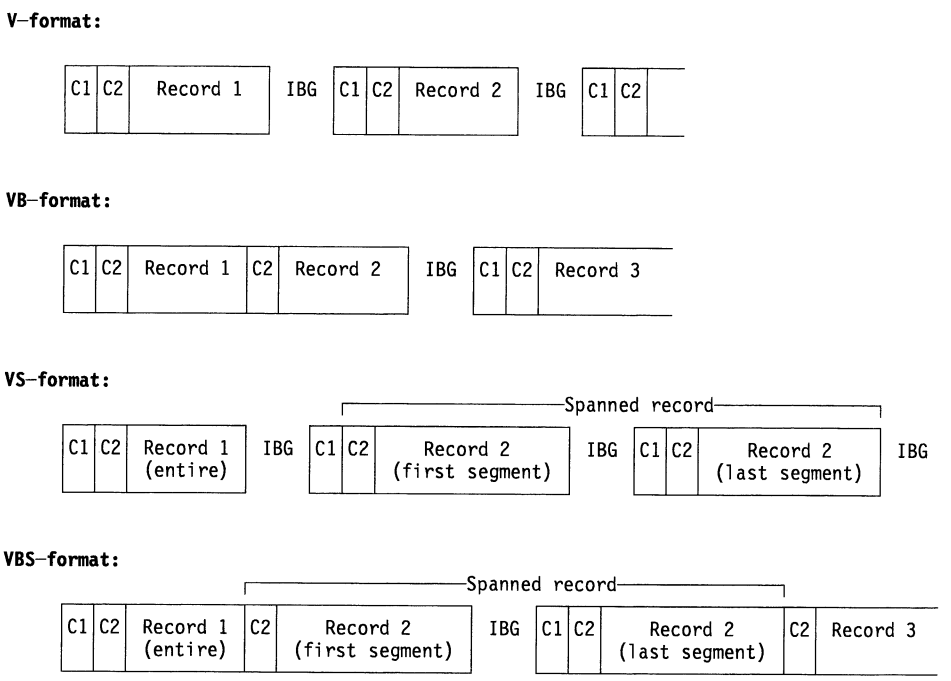
You can specify the following formats for variable-length records:

V	Variable-length, unblocked
VB	Variable-length, blocked
VS	Variable-length, unblocked, spanned
VBS	Variable-length, blocked, spanned
D	Variable-length, unblocked, ASCII
DB	Variable-length, blocked, ASCII.

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). Illustrations of variable-length records are shown in Figure 10 on page 72.

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.



C1: Block control information
C2: Record or segment control information

Figure 10. Variable-length records

Spanned records: A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If this occurs, the record is divided into segments and accommodated in two or more consecutive blocks by specifying the record format as either VS or VBS. Segmentation and reassembly are handled by the operating system. The use of spanned records allows you to select a block size, independently of record length, that will provide optimum use of auxiliary storage.

VS-format is similar to V-format. Each block contains only one record or segment of a record. The first 4 bytes of the block contain block control information, and the next 4 contain record or segment control information (including an indication of whether the record is complete or is a first, intermediate, or last segment).

With REGIONAL(3) organization, the use of VS-format removes the limitations on block size imposed by the physical characteristics of the direct-access device. If the record length exceeds the size of a track, or if there is no room left on the current track for the record, the record will be spanned over one or more tracks.

VBS-format differs from VS-format in that each block contains as many complete records or segments as it can accommodate; each block is, therefore, approximately the same size (although there can be a variation of up to 4 bytes, since each segment must contain at least 1 byte of data).

Note: You should exercise caution when reading spanned records with an INPUT file. PL/I uses your RECSIZE value to allocate buffers to read the file, but it may be possible that the actual records on the data set are longer than the RECSIZE value. If this is the case, the extra data in the records could overwrite some storage in your program's partition. PL/I will attempt to cancel the program if this happens, but the extra data could have overwritten some important program control information. In this case, the results are unpredictable.

ASCII records: For data sets that are recorded in ASCII, use D-format as follows:

- D-format records are similar to V-format, except that the data they contain is recorded in ASCII.
- DB-format records are similar to VB-format, except that the data they contain is recorded in ASCII.

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 two main types of non-VSAM data sets and the corresponding keywords describing their PL/I organization[2] are as follows:

Type of data set	PL/I organization
Sequential	CONSECUTIVE
Direct	REGIONAL

Note: A third data set type, Indexed Sequential, is no longer supported. If you specify INDEXED as the file organization, PL/I assumes the data set is VSAM.

PL/I also provides support for four types of VSAM data organization: *ESDS*, *KSDS*, *RRDS*, and *VRDS*. For more information about VSAM data sets, see Chapter 8, "Defining and using VSAM data sets" on page 134.

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 is used for all magnetic tapes, and can be selected for direct-access devices.

A *direct* (or REGIONAL) data set must reside on a direct-access volume. The records within the data set are placed within *regions*, and can be organized in one of three ways:

- REGIONAL(1) data sets contain one record per region, and the records are identified by their region number. The records are fixed length.
- REGIONAL(2) data sets contain fixed-length records, identified by a combination of their region number and a recorded key.
- REGIONAL(3) data sets can contain one or more records per region, and the records are identified by a combination of their region number and recorded key. The records can be fixed, variable, or undefined length.

Sequential processing is also possible with regional data sets.

---

[2] 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.

Labels

The operating system uses internal labels to identify magnetic-tape and direct-access volumes, and to identify the data sets held on those volumes.

Magnetic-tape volumes can have IBM standard or nonstandard labels, or they can be unlabeled. IBM standard labels have two parts: the initial volume label, and header and trailer labels. The initial volume label identifies a volume and its owner; the header and trailer 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. Trailer labels are almost identical with header labels, and are used when magnetic tape is read backward.

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, called a *data set label* or *file label*, for each data set stored on the volume.

Job control statements for data sets

The data set information supplied in the ENVIRONMENT attribute includes the data set organization, the record format, and the type of input/output device that will be used. But it does not include the actual device address or the name of the data set; that information, if required, must always be given in VSE/ESA job control statements. This arrangement permits PL/I programs to be written without knowledge of the actual data sets to be processed, and enables the same program to be used to process different data sets of a similar type.

The VSE/ESA job control statements are:

ASSGN identifies the device on which the data set will be mounted.

TLBL labeled magnetic tape only; identifies the data set.

DLBL direct access devices only; identifies the data set.

EXTENT direct access devices only; identifies the location of the data set.

These statements are discussed below. For more detailed information, refer to *VSE/ESA System Control Statements.*

The ASSGN statement

The VSE/ESA operating system uses *symbolic device names* to identify the physical devices that contain data sets. The MEDIUM option of the PL/I ENVIRONMENT attribute specifies the same symbolic device name. It is the function of the ASSGN statement to relate this symbolic name to a physical device address; such a relationship is called a *device assignment.*

Many device assignments are established permanently when the VSE/ESA system is initialized (IPLed). These are *standard device assignments.* If a program uses a standard device assignment for a data set, an ASSGN statement is not required for that data set. Therefore, you should always be aware of the standard assignments at your installation before running a program that processes a data set.

The TLBL statement

A program must include a TLBL statement for each magnetic tape data set that has IBM standard labels. The statement is not required for magnetic tape data sets that have non-standard labels or are unlabeled.

The TLBL statement contains information that identifies the data set (including the data set name and the volume serial number); this information is recorded in the data set labels.

The DLBL and EXTENT statements

For each data set on a direct access device or diskette, there must be a DLBL statement and, optionally, one or more EXTENT statements.

The DLBL statement, like the TLBL statement for magnetic tape, contains information that identifies the data set.

For a direct access device, the EXTENT statement defines the starting location and size of the data set. The statement is not required for existing single-volume data sets accessed for input or update where the MEDIUM option has been specified in the file declaration. It is also not required for VSAM data sets.

For a diskette, an EXTENT statement defines the type of extent; only data areas (signified by 1 in the EXTENT statement) are supported.

VSE/VSAM space management for SAM data sets

You can define sequential data sets in VSAM space by using the VSE/VSAM Space Management for SAM feature. This applies to CONSECUTIVE RECORD files and STREAM files where the data set resides on a DASD device.

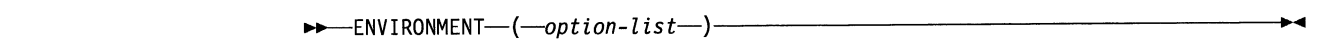
The data set definition can be explicit or implicit:

- For *explicit* defining, use the Access Method Services utility program to define a SAM ESDS with the required RECORDSIZE and RECORDFORMAT. Then in the JCL to run your program, supply a DLBL statement specifying VSAM. You do not need an EXTENT statement.
- For *implicit* defining, supply a DLBL statement specifying VSAM, and the RECORDS and RECSIZE parameters. You can specify the volume with an EXTENT statement, or use a default model for a SAM ESDS.

For CONSECUTIVE INPUT files and for CONSECUTIVE UNBUFFERED files that are to be opened for both OUTPUT and INPUT, specify DISP=OLD on the DLBL statement.

The ENVIRONMENT attribute

You use the ENVIRONMENT attribute of a PL/I file declaration 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.



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 8, "Defining and using VSAM data sets" on page 134).

```
DCL FILENAME FILE RECORD SEQUENTIAL
    INPUT ENV(VSAM GENKEY);
```

Table 9 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 9 (Page 1 of 2). Attributes of PL/I file declarations

Data set type	Stream	Record							Legend: C Checked for VSAM; D Default; I Must be specified or implied; N Ignored for VSAM; O Optional; S Must be specified; x Syntax-checked and ignored; - Invalid
		Sequential					Direct		
		Consecutive		Regional					
File Type	Consecutive	Buffered	Unbuffered	Buffered	Unbuffered	VSAM	Regional	VSAM	
File attributes[1]									**Attributes implied**
File	I	I	I	I	I	I	I	I	
Input[1]	D	D	D	D	D	D	D	D	File
Output	O	O	O	O	O	O	O	O	File
Environment	I	I	I	S	S	S	S	S	File
Stream	D	-	-	-	-	-	-	-	File
Print[1]	O	-	-	-	-	-	-	-	File stream output
Record	-	I	I	I	I	I	I	I	File
Update[2]	-	O	O	O	O	O	O	O	File record
Sequential	-	D	D	D	D	D	-	-	File record
Buffered	-	D	-	D	-	D	-	S	File record
Unbuffered	-	-	S	-	S	S	D	D	File record
Backwards[3]	-	O	O	-	-	-	-	-	File record sequential input
Keyed[4]	-	-	-	O	O	O	I	O	File record
Direct	-	-	-	-	-	-	S	S	File record keyed
Exclusive[5]	-	-	-	-	-	-	x	-	
ENVIRONMENT options for record formats and sizes									**Comments**
F|FB|V|VB| VS|VBS|U	I	S	S	-	-	N	-	N	VS and VBS are invalid with STREAM
F|FB|D|DB|U	S	S	-	-	-	N	-	N	ASCII data sets only
F|V|VS|U	-	-	-	S	S	N	S	N	Only F for REGIONAL(1) and (2)
F|FB|V|VB	-	-	-	-	-	N	-	N	VS invalid with UNBUFFERED
RECSIZE(n)	I	I	I	I	I	C	I	C	RECSIZE and/or BLKSIZE must be specified
BLKSIZE(n)	I	I	I	I	I	N	I	N	for consecutive and regional files

Table 9 (Page 2 of 2). Attributes of PL/I file declarations

Data set type	Stream	Record							Legend: C Checked for VSAM; D Default; I Must be specified or Implied; N Ignored for VSAM; O Optional; S Must be specified; x Syntax-checked and ignored; - Invalid
		Sequential					Direct		
		Consecutive		Regional					
File Type	Consecutive	Buffered	Unbuffered	Buffered	Unbuffered	VSAM	Regional	VSAM	
Other ENVIRONMENT options (arranged alphabetically)									**Comments**
ADDBUFF[6]	-	-	-	-	-	-	-	-	
ASCII	O	O	-	-	-	-	-	-	
BKWD	-	-	-	-	-	O	-	-	
BUFFERS(n)	I	I	-	I	-	N	-	N	
BUFND(n)	-	-	-	-	-	O	-	O	
BUFNI(n)	-	-	-	-	-	O	-	O	
BUFOFF(n)	O	O	-	-	-	-	-	-	
BUFSP(n)	-	-	-	-	-	O	-	O	
CMDCHN(n)	O	O	O	-	-	-	-	-	
COBOL	-	O	O	O	O	O	O	O	
COMPAT	-	-	-	O	O	-	O	-	For REGIONAL(3) fixed format only
CONSECUTIVE	D	D	D	-	-	O	-	O	Allowed for VSAM ESDS
CTLASA|CTL360	-	O	O	-	-	-	-	-	Invalid for ASCII data sets
DSN(n)	-	-	-	-	-	O	-	O	
EXTENTNUMBER(n)[7]	-	-	-	x	x	-	x	-	
FILESEC	I	I	I	-	-	-	-	-	
GENKEY	-	-	-	-	-	O	-	-	INPUT or UPDATE files only: KEYED is required
GRAPHIC	O	-	-	-	-	-	-	-	
HIGHINDEX[6]	-	-	-	-	-	-	-	-	
INDEXAREA(n)[6]	-	-	-	-	-	-	-	-	
INDEXED	-	-	-	-	-	O	-	O	Treated by PL/I VSE as VSAM KSDS
INDEXMULTIPLE[6]	-	-	-	-	-	-	-	-	
KEYLENGTH(n)	-	-	-	S	S	C	S	C	For REGIONAL(2) and (3) OUTPUT only
KEYLOC(n)[6]	-	-	-	-	-	-	-	-	
LEAVE	O	O	O	-	-	-	-	-	
LIMCT	-	-	-	-	-	-	O	-	For REGIONAL(2) and REGIONAL(3) only, when COMPAT is not used
MEDIUM(device)[8]	O	O	O	O	O	N	O	N	
NOFEED	O	O	O	-	-	-	-	-	
NOLABEL[7]	x	x	x	-	-	-	-	-	
NOTAPEMK	O	O	-	-	-	-	-	-	
NOWRITE[6]	-	-	-	-	-	-	-	-	
OFLTRACKS[6]	-	-	-	-	-	-	-	-	
PASSWORD	-	-	-	-	-	O	-	O	
REGIONAL({1|2|3})	-	-	-	S	S	-	S	-	
REREAD	O	O	O	-	-	-	-	-	
REUSE	-	-	-	-	-	O	-	O	OUTPUT file only
SCALARVARYING	-	O	O	O	O	O	O	O	Invalid for ASCII data sets
SIS[5]	-	-	-	-	-	x	-	x	
SKIP	-	-	-	-	-	O	-	-	
TOTAL	-	O	-	-	-	-	-	-	
TRKOFL[5]	-	-	-	-	-	-	-	-	
UNLOAD	O	O	O	-	-	-	-	-	
VERIFY	O	O	O	O	O	-	O	-	
VOLSEQ	O	O	O	-	-	-	-	-	
VSAM	-	-	-	-	-	S	-	S	VSAM is not required if INDEXED is specified
WRTPROT	O	O	O	-	-	-	-	-	

Notes:

1. A file with the INPUT attribute cannot have the PRINT attribute.
2. UPDATE is invalid for tape files.
3. BACKWARDS is valid only for input tape files.
4. Keyed is required for REGIONAL output.
5. Syntax checked only; has no effect. Kept for source compatibility with PL/I MVS & VM.
6. Syntax checked only; has no effect. Kept for source compatibility with previous releases of PL/I. These options apply to indexed sequential files, which are not supported. PL/I VSE treats indexed sequential file definitions as VSAM.
7. Syntax checked only; has no effect. Kept for source compatibility with the DOS PL/I Optimizing Compiler.
8. For DASD data sets, the logical unit number can be specified either in the MEDIUM option or on the EXTENT JCL statement.

Data set organization options

The options that specify data set organization are:

```
CONSECUTIVE
REGIONAL({1 | 2 | 3})
VSAM
```

Each is described in the discussion of the data set organization to which it applies.

If you don't specify the data set organization option in the ENVIRONMENT attribute or in the OPEN statement, the default is CONSECUTIVE.

Note: INDEXED is also permitted as a data set organization option. PL/I treats this as VSAM (see "Using files defined for non-VSAM data sets" on page 149).

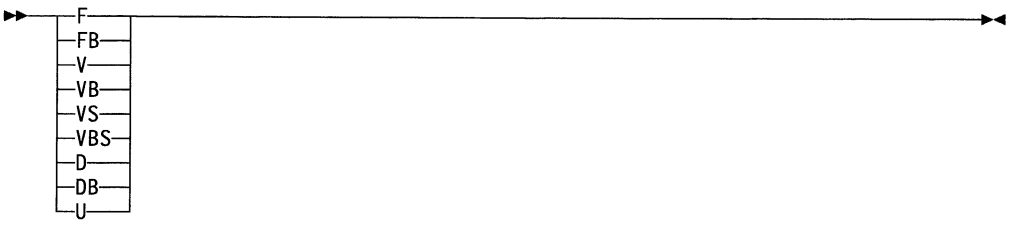
Other ENVIRONMENT options

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.

For consecutive data sets on CKD DASD devices, you can also specify the BLKSIZE option on the DLBL JCL statement, provided that TOTAL is not specified as an ENVIRONMENT option. For details, see the *VSE/ESA System Control Statements* book.

Record formats for record-oriented data transmission

Record formats supported depend on the data set organization.



Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
Variable-length	V	unblocked
	VB	blocked
	VS	spanned
	VBS	blocked, spanned
	D	unblocked, ASCII
	DB	blocked, ASCII
Undefined-length	U	(cannot be blocked)

Note: Fixed-length formats FS and FBS will also be accepted by the compiler for compatibility with the MVS implementation. These will be treated as F and FB respectively.

When U-format records are read into a varying-length string, PL/I sets the length of the string to the block length of the retrieved data.

These record format options do not apply to VSAM data sets. If you specify a record format option for a file associated with a VSAM data set, the option is ignored.

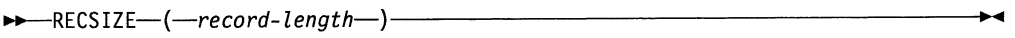
You can only specify VS-format records for data sets with consecutive or REGIONAL(3) organization.

Record formats for stream-oriented data transmission

The record format options for stream-oriented data transmission are discussed in "Using stream-oriented data transmission" on page 86.

RECSIZE option

The RECSIZE option specifies the record length.



For files other than those associated with VSAM data sets, **record-length** is the sum of:

1. The length required for data. For variable-length and undefined-length records, this is the maximum length.
2. Any control bytes required. Variable-length records require 4 (for the record-length prefix); fixed-length and undefined-length records do not require any.

For VSAM data sets, the maximum and average lengths of the records are specified to the Access Method Services utility when the data set is defined. If you include the RECSIZE option in the file declaration for checking purposes, you should specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

You can specify **record-length** as an integer or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:

Fixed-length, and undefined (except ASCII data sets): 32760

V-format, and VS- and VBS-format with UPDATE files: 32756

VS- and VBS-format with INPUT and OUTPUT files: 16 777 215 (that is, 16 megabytes)

ASCII data sets: 9999

VSAM data sets: 32761 for unspanned records. For spanned records, the maximum is the size of the control area.

Zero value:

Default action is taken (see "Record format, BLKSIZE, and RECSIZE defaults" on page 81).

Negative Value:

The UNDEFINEDFILE condition is raised.

BLKSIZE option

The BLKSIZE option specifies the maximum block size on the data set.

block-size is the sum of:

1. The total length(s) of one of the following:
   - A single record
   - A single record and either one or two record segments
   - Several records
   - Several records and either one or two record segments
   - Two record segments
   - A single record segment.

   For variable-length records, the length of each record or record segment includes the 4 control bytes for the record or segment length.

   The above list summarizes all the possible combinations of records and record segments options: fixed- or variable-length, blocked or unblocked, spanned or unspanned. When specifying a block size for spanned records, you must be aware that each record and each record segment requires 4 control bytes for the record length, and that these quantities are in addition to the 4 control bytes required for each block.
2. Any further control bytes required.
   - Variable-length blocked records require 4 (for the block size).
   - Fixed-length and undefined-length records do not require any further control bytes.
3. Any block prefix bytes required (ASCII data sets only).

block-size can be specified as an integer, or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:
32760 (or 9999 for an ASCII data set for which BUFOFF without a prefix-length value has been specified).

Zero value:
If you set BLKSIZE to 0, depending on what software is installed on your system, the block size may be supplied at run time from the JCL. For an elaboration of this topic, see "Record format, BLKSIZE, and RECSIZE defaults" on page 81.

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, including the 4 byte prefix
2. Four control bytes.

V-format:
The block size must be at least equal to the record length plus 4.

VS-format or VBS-format:
The block size can be less than, equal to, or greater than the record length.

DB-format:
The block size must be equal to or greater than the sum of:

1. The maximum length of any record
2. The length of the block prefix (if block is prefixed).

Notes:

- For unblocked F-format records, specify BLKSIZE but not RECSIZE, and use a record format of F.
- For unblocked V- or D-format records, specify both BLKSIZE and RECSIZE, and use a record format of V or D. If you use VB or DB, the system might block some of the records if the actual record length is short enough.
- If for FB-format records the block size equals the record length, the record format is set to F.
- For REGIONAL(3) data sets with VS format, record length cannot be greater than block size.
- The BLKSIZE option does not apply to VSAM data sets, and is ignored if you specify it.

Record format, BLKSIZE, and RECSIZE defaults

If you do not specify the record format, block size, or record length for a non-VSAM data set, PL/I attempts to resolve these values when the data set is opened.

For data sets controlled by the VSE/VSAM Space Management for SAM feature, the information in the VSAM catalog is not reliable and therefore is not used to resolve record format, block size or record length. If your installation uses special software to manage disk and/or tape files, PL/I attempts to obtain the information from that software.

If your installation does not have this software, or PL/I is unable to obtain the information from that software, the following default action is taken:

Record format:
The record format is set to F.

Block size:
For a blocked file, if the DLBL statement specifies a block size, this value is used. If it is incompatible with the RECSIZE value, the UNDEFINEDFILE condition is raised. If the DLBL statement does not specify a block size, a value is derived from the RECSIZE option (with the addition of any control or prefix bytes).

For an unblocked file, a block size value is derived from the RECSIZE option (with the addition of any control or prefix bytes).

Record length:

For a blocked file, if the record length is not specified, the UNDEFINEDFILE condition is raised.

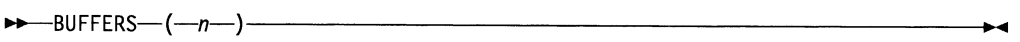
For an unblocked file, the record length is derived from the BLKSIZE specification (with the subtraction of any control or prefix bytes).

If neither record length nor block size is specified, the UNDEFINEDFILE condition is raised.

BUFFERS option

A buffer is a storage area that is used for the intermediate storage of data transmitted to and from a data set. The use of buffers can speed up processing of SEQUENTIAL files. Buffers are essential for blocking and deblocking records and for locate-mode transmission.

Use the BUFFERS option in the ENVIRONMENT attribute to specify buffers to be allocated for CONSECUTIVE and REGIONAL data sets accessed sequentially, according to the following syntax:



where **n** is the number of buffers you want allocated for your file.

The actual number of buffers used will only ever be one or two. If you specify any value other than one, two buffers will be used.

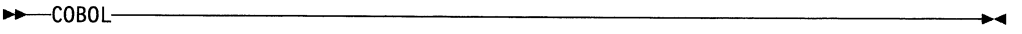
PL/I allocates the following number of buffers for these data sets, regardless of the value you specify:

- A REGIONAL data set is always allocated one buffer.
- A card reader data set is always allocated two buffers.
- A printer data set is always allocated two buffers.

The BUFFERS option is ignored for VSAM; you use the BUFNI, BUFND, and BUFSP options instead.

COBOL option — data interchange

The COBOL option specifies that structures in the data set associated with the file will be mapped as they would be in a COBOL compiler. The COBOL structures can be synchronized or unsynchronized; it is your responsibility to ensure that the associated PL/I structure has the equivalent alignment stringency; that is, it must be ALIGNED or UNALIGNED, respectively.

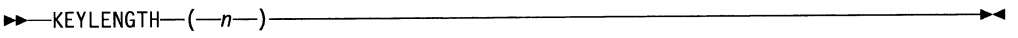


The following restrictions apply to the handling of a file with the COBOL option:

- You can only use a file with the COBOL option for READ INTO, WRITE FROM, and REWRITE FROM statements.
- You cannot pass the file name as an argument or assign it to a file variable.
- You must subscript any array variable to be transmitted.
- If a condition is raised during the execution of a READ statement, you cannot use the variable named in the INTO option in the ON-unit. If the completed INTO variable is required, there must be a normal return from the ON-unit.
- You can use the EVENT option only if the compiler determines that the PL/I and COBOL structure mappings are identical (that is, all elementary items have identical boundaries). If the mappings are not identical, or if the compiler cannot tell whether they are identical, an intermediate variable is created to represent the level-1 item as mapped by the COBOL algorithm. The PL/I variable is assigned to the intermediate variable before a WRITE statement is executed, or assigned from it after a READ statement has been executed.

KEYLENGTH option

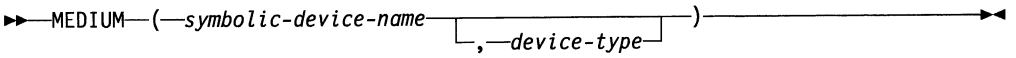
Use the KEYLENGTH option to specify the length of the recorded key for KEYED files where *n* is the length. You can specify KEYLENGTH for REGIONAL(2) and REGIONAL(3) files.



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.

MEDIUM option — device specification

The MEDIUM option tells the compiler the logical device name of the device that contains the data set.



symbolic-device-name
The logical device assignment of the device that contains the data set. It has the form SYS*xxx*, where *xxx* may be:

IPT	The system input device.
LST	The system output device (printer).
PCH	The system output device (card punch).
000-254	Symbolic device names SYS000 through SYS254.

device-type
This is optional, and is included for compatibility with the DOS PL/I Optimizing Compiler. If you include it, it will be ignored.

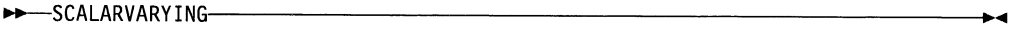
MEDIUM should not be specified, and is ignored, for VSAM files. For non-VSAM files, if MEDIUM is not specified, the data set is assumed to reside on a DASD

device, and the symbolic device name must then be supplied at run time with an EXTENT JCL statement.

You associate the symbolic device name with a real device at run time by means of a JCL ASSGN statement. This is described in "Job control statements for data sets" on page 74.

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.



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.

VERIFY option — data checking

The VERIFY option specifies that, after every physical write operation, the system will perform a read-check to validate the write.

This option is only permitted for files associated with direct-access devices.

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 10 on page 85 shows the facilities that are available with the various types of data sets that can be used with PL/I Record I/O.

The following chapters describe how to use Record I/O data sets for different types of data sets:

- Chapter 6, "Defining and using consecutive data sets" on page 86
- Chapter 7, "Defining and using regional data sets" on page 115
- Chapter 8, "Defining and using VSAM data sets" on page 134

Table 10. A comparison of data set types available to PL/I record I/O

	VSAM KSDS	**VSAM ESDS**	**VSAM RRDS and VRDS**	**CONSECUTIVE**	**REGIONAL (1)**	**REGIONAL (2)**	**REGIONAL (3)**
Sequence	Key order	Entry order	Numbered	Entry order	By region	By region	By region
Devices	DASD	DASD	DASD	DASD, tape, card, etc.	DASD	DASD	DASD
Access							
1 By key	1	1	1		1	1	1
2 Sequential	2	2	2	2	2	2	2
3 Backward	3	3	3	3 tape only			
Alternate index access			No	No	No	No	No
1 By key	1	1					
2 Sequential	2	2					
3 Backward	3	3					
How extended	With new keys	At end	In empty slots	At end	In empty slots	With new keys	With new keys
Spanned records	Yes	Yes	No	Yes	No	No	Yes
Deletion	Yes	No	Yes	No	Yes	Yes	Yes
1 Space reusable	1		1		1	1	
2 Space not reusable							2

Chapter 6. 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, or, for RECORD I/O only, also in the reverse order when using the BACKWARDS attribute. See Table 9 on page 76 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 JCL 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 VSE 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 9 on page 76; the FILE attribute is described in the *PL/I VSE Language Reference.* The PRINT attribute is described further in "Using PRINT files with stream I/O" on page 94. Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

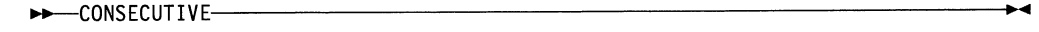
Table 9 on page 76 summarizes the ENVIRONMENT options. The options applicable to stream-oriented data transmission are listed here.

```
CONSECUTIVE
F|FB|V|VB|D|DB|U
RECSIZE(record-length)
BLKSIZE(block-size)
MEDIUM(SYSxxx)
BUFFERS(n)
GRAPHIC

ASCII
BUFOFF[(n)]
CMDCHN(n)
CTLASA|CTL360
FILESEC
LEAVE|REREAD|UNLOAD
NOFEED
NOTAPEMK
VOLSEQ
WRTPROT
```

For the options above the blank line, BLKSIZE and BUFFERS are described in Chapter 5, "Using data sets and files," beginning on page 80, and the remaining options are described immediately below. The options below the blank line are common to both STREAM and RECORD I/O, and are described later in this chapter, beginning on page 101.

CONSECUTIVE

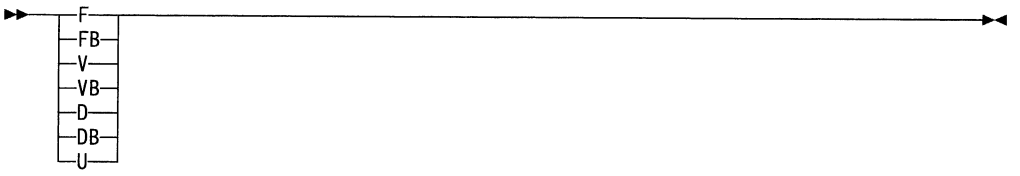
STREAM files must have CONSECUTIVE data set organization; however, it is not necessary to specify this in the ENVIRONMENT options since CONSECUTIVE is the default data set organization. The CONSECUTIVE option for STREAM files is the same as that described in "Data set organization" on page 73.



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.



Records can have one of the following formats, which are described in "Record formats" on page 70.

Fixed-length	F	unblocked
	FB	blocked
Variable-length	V	unblocked
	VB	blocked
	D	unblocked ASCII
	DB	blocked ASCII
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 "The ENVIRONMENT attribute" on page 75. 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 VSE Language Reference.*

Additional record-size considerations for list- and data-directed transmission of graphics are given in the *PL/I VSE 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, 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 81.

Output files:

Record format:

Set to F-format, or if ASCII option specified, to D-format.

Record length:
The specified or default LINESIZE value is used:

PRINT files:

F, FB, or U:	line size + 1
V, VB, D, or DB:	line size + 5

Non-PRINT files:

F, FB, or U:	linesize
V, VB, D, or DB:	linesize + 4

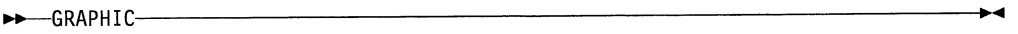
Block size:

F or FB:	record length
V or VB:	record length + 4
D or DB:	record length + block prefix

(see "Information interchange codes" on page 70)

GRAPHIC option

You must specify the GRAPHIC option of the ENVIRONMENT attribute if you use DBCS variables or DBCS constants in GET and PUT statements for list- and data-directed I/O. You can also specify the GRAPHIC option for edit-directed I/O.



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 VSE Language Reference.*

Creating a data set with stream I/O

To create a data set, you must give the operating system certain information in your PL/I program, and in the JCL statements that define the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

Essential information

You must supply the following information, summarized in Table 11 on page 90, when creating a data set:

- The symbolic device name of the device that will write your data set (SYS*nnn*). You can specify this in the MEDIUM ENVIRONMENT option, or for DASD data sets, on the EXTENT JCL statement.

- The record format. If you do not specify a record format, F-format is the default.
- The record size.
- The block size if the records are blocked.

If you want your data set stored on a direct-access device, you must also specify:

- The name of the data set (on the DLBL statement).
- The name of the direct access volume that will contain the data set (on the EXTENT statement).
- The physical location of the data set on that volume. This is the starting track number and number of tracks, specified on the EXTENT statement.

For data sets on magnetic tape, you must also specify:

- The name of the data set (on the TLBL statement).
- The ID of the tape volume that will contain the data set. This is specified in the 'file-serial-number' option of the TLBL statement.
- If your data set is not the first (or only) data set on a magnetic-tape volume, you must use the 'volume-sequence-number' option of the TLBL statement to indicate its sequence number on the tape.

Table 11. Creating a data set with stream I/O: essential information

Storage device	When required	What you must state	Where specified
All	Always	Symbolic device name	File declaration in source program (MEDIUM(SYS...)) or EXTENT statement for DASD data sets[1]
		Record format	File declaration (F|FB|V|VB|D|DB|U)
		Record size	File declaration (RECSIZE)
All	Non-standard device assignment	Device assignment	ASSGN statement
Magnetic tape only	Standard labelled tape file	Data set name	TLBL statement
Direct access data sets	Always	Data set name	DLBL statement
		Data set extent	EXTENT statement[2]

Notes:

1. The EXTENT statement overrides the MEDIUM option if both are specified.
2. The EXTENT statement is not always required when using VSE/VSAM Space Management for SAM.

Examples

Figure 11 on page 91 shows an example of a program that uses edit-directed stream-oriented data transmission to create a data set on a direct access storage device. 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.

```
// JOB    EX7#2
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM OUTPUT ENVIRONMENT(U),
             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;
/*
// EXEC   LNKEDT
// ASSGN  SYS009,3390,VOL=VSE222,SHR
// DLBL   WORK,'HPU8.PEOPLE',0,SD
// EXTENT SYS009,VSE222,1,0,3450,15
// EXEC   ,SIZE=128K
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
K.J.MANSON         4 401539 PROGRAMMER       LEE    SUZY   KATHIE JENNI
/*
/&
```

Figure 11. Creating a data set with stream-oriented data transmission

Figure 12 on page 92 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.

```
// JOB    EX7#3
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
% PROCESS GRAPHIC;
  XAMPLE1:  PROC OPTIONS(MAIN);
            DCL  INFILE FILE INPUT RECORD,
                 OUTFILE FILE OUTPUT STREAM ENV(MEDIUM(SYS010) 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 UP  */
               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;
/*
// EXEC   LNKEDT
// ASSGN  SYS010,SYSLST
// EXEC   ,SIZE=128K
ABCDEF<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 12. 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 identify it to the operating system in your JCL. The following paragraphs describe the essential information you must include in the JCL statements or in your program source, 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 accessing a data set using stream-oriented transmission, you must specify:

- The symbolic device name. You can specify this in the MEDIUM ENVIRONMENT option, or for DASD data sets, in the EXTENT JCL statement.
- The record format, record length, and if records are blocked, the block size. You normally specify these as ENVIRONMENT options in the file declaration, but depending on the software level at your installation, the system may be able to supply this information at run time from the JCL parameters.
- For standard labelled data sets on tape or direct access devices, the data set name. You specify this on the TLBL or DLBL JCL statements.

If the data set follows another data set on a magnetic-tape volume, you must use the 'file-sequence-number' parameter of the TLBL statement to indicate its relative position on the tape.

Magnetic tape without IBM standard labels

If a magnetic tape data set has nonstandard labels or is unlabeled, you must indicate this by omitting the TLBL statement from your JCL. The ASSGN statement matches the MEDIUM option and points to the device, and the absence of a TLBL statement indicates that the tape on the device does not have standard labels.

PL/I has no facilities for processing nonstandard labels. These appear to the operating system, and to PL/I, as other data sets preceding or following your data set. You can either process the labels as independent data sets or use the MTC JCL statement to bypass them.

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 13 reads the data set created by the program in Figure 11 on page 91 and uses the file SYSPRINT to list the data it contains. (For details on SYSPRINT, see "Using SYSIN and SYSPRINT files" on page 99.) 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.

```
// JOB    EX7#5
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM INPUT ENVIRONMENT(U RECSIZE(400)),
             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;
/*
// EXEC   LNKEDT
// ASSGN  SYS009,DISK,VOL=VSE222,SHR
// DLBL   WORK,'HPU8.PEOPLE'
// EXTENT SYS009
// EXEC   ,SIZE=128K
/&
```

Figure 13. 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 Table 13 on page 104 and Table 15 on page 105.)

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 magnetic-tape or 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.

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 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 magnetic-tape or 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.

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 F-format records. The complete default information is:

```
ENVIRONMENT(F RECSIZE(121) BLKSIZE(121) BUFFERS(2))
```

Example: Figure 14 on page 96 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 compiler infers the following from the declaration of the file TABLE, and the line size specified in the statement that opens it:

Record format = F
(the default for a STREAM file).

Record size = 94
(line size + 1 byte for print control character).

Block size = 94
(same as record length for F-format files).

The program in Figure 19 on page 114 uses record-oriented data transmission to print the table created by the program in Figure 14.

```
// JOB     EX7#6
// OPTION  LINK
// EXEC    IEL1AA,SIZE=128K
*PROCESS INT F(I) AG A(F) ESD MAP 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;
/*
// EXEC   LNKEDT
// DLBL   TABLE,'SINE.LIST',0,SD
// EXTENT SYS011,VSE111,1,0,3450,30
// ASSGN  SYS011,DISK,VOL=VSE111,SHR
// EXEC   ,SIZE=128K
/&
```

Figure 14. Creating a print file via stream data transmission. The example in Figure 19 on page 114 will print the resultant file.

The tab control table

The tab control table controls the default alignment of output to PRINT files. It defines the number of lines per page, and for data-directed and list-directed output, the preset tabulator positions (columns) where each field will be printed. The tab table contains the following fields:

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 is the number of lines that will be printed on each page before the ENDPAGE condition is raised. This value is also used for dump output to the PLIDUMP data set.

LINESIZE:
: Halfword binary integer that defines the default line size. This is the number of character positions that can be printed on each line.

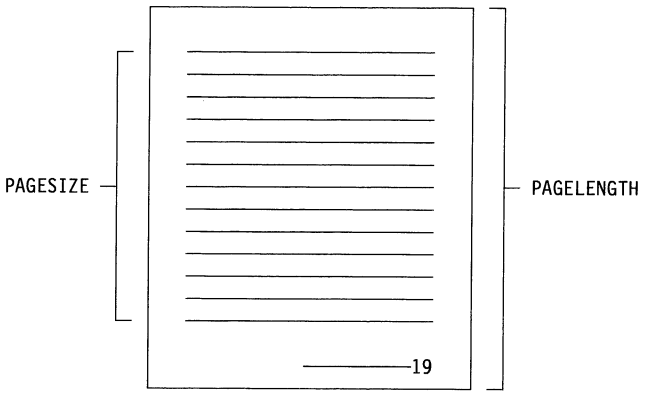
PAGELENGTH:
: Halfword binary integer that defines the default page length. This is the physical page length, that is, the number of lines that can fit onto the physical page. Figure 15 on page 98 shows the relationship between PAGESIZE and PAGELENGTH.

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.

Figure 16 on page 98 shows a tab table with the default settings.



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.

```
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(5),
      TAB1 INIT(25),
      TAB2 INIT(49),
      TAB3 INIT(73),
      TAB4 INIT(97),
      TAB5 INIT(121)) FIXED BIN(15,0);
```

Figure 16. PL/I structure and default settings for the tab control table

Overriding the tab control table: You can override the PAGESIZE and LINESIZE settings in the OPEN statement for PRINT files. See the *PL/I VSE Language Reference* for the syntax of the OPEN statement.

You can override the default PL/I tab settings for all of the PRINT files in your program by creating a tab table in the format described above, and using it to override the default table.

There are two methods of supplying the overriding tab table. One method is to include a PL/I structure in your source program with the name PLITABS, which you must declare to be STATIC EXTERNAL. An example of the PL/I structure is shown in Figure 17 on page 99. 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.

The second method is to create an assembler language control section named PLITABS, equivalent to the structure shown in Figure 17, and to include it when link-editing your PL/I program.

```
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 17. PL/I structure PLITABS for modifying the preset tab settings

Using SYSIN and SYSPRINT files

If you code a GET statement without the FILE or STRING option in your program, the compiler inserts the file name SYSIN. If you code a PUT statement without the FILE or STRING 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
     ENV(MEDIUM(SYSLST) F RECSIZE(121) BLKSIZE(121) BUFFERS(2))
```

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 LE/VSE message file option, see the *LE/VSE 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 data set information in your JCL.

Using record-oriented data transmission

PL/I supports various types of data sets with the RECORD attribute (see Table 20 on page 109). This section covers how to use consecutive data sets.

Table 12 on page 100 lists the statements and options that you can use to create and access a consecutive data set using record-oriented data transmission.

Table 12. 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)

Table 12. 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 UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED[3]	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED[3]	READ FILE(file-reference) INPUT(reference);	EVENT(event-reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-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);
3. You can specify the BACKWARDS attribute for files on magnetic tape.

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 | UNBUFFERED
            [BACKWARDS]
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 9 on page 76. The file attributes are described in the *PL/I VSE Language Reference.* Options of the ENVIRONMENT attribute are discussed below.

Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to consecutive data sets are:

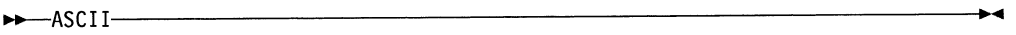
```
F|FB|V|VB|VS|VBS|D|DB|U
RECSIZE(record-length)
BLKSIZE(block-size)
MEDIUM(SYSxxx)
SCALARVARYING
COBOL
BUFFERS(n)

ASCII
BUFOFF[(n)]
CMDCHN(n)
CONSECUTIVE
CTLASA|CTL360
FILESEC
LEAVE|REREAD|UNLOAD
NOFEED
NOTAPEMK
TOTAL
VERIFY
VOLSEQ
WRTPROT
```

The options above the blank line are described in "The ENVIRONMENT attribute" on page 75, and those below the blank line are described below. D- and DB-format records are also described below.

See Table 9 on page 76 to find which options you must specify, which are optional, and which are defaults.

ASCII

The ASCII option specifies that the code used to represent data on the data set is ASCII.



You can create and access data sets on magnetic tape using ASCII in PL/I. The implementation supports F, FB, U, D, and DB record formats. F, FB, and U formats are treated in the same way as other data sets; D and DB formats, which correspond to V and VB formats in other data sets, are described below.

Only character data can be written to an ASCII data set; therefore, when you create the data set, you must transmit your data from character variables. You can give these variables the attribute VARYING as well as CHARACTER, but you cannot transmit the two length bytes of varying-length character strings. In other words, you cannot use a SCALARVARYING file to transmit varying-length character strings to an ASCII data set. Also, you cannot transmit data aggregates containing varying-length strings.

Since an ASCII data set must be on magnetic tape, it must be of consecutive organization. The associated file must be BUFFERED. You can also specify the BUFOFF ENVIRONMENT option for ASCII data sets.

If you do not specify ASCII in the ENVIRONMENT option, but you specify BUFOFF, D, or DB, then ASCII is the default.

D-Format and DB-format records: The data contained in D- and DB-format records is recorded in ASCII. Each record can be of a different length. The two formats are:

D-format:
The records are unblocked; each record constitutes a single block. Each record consists of:

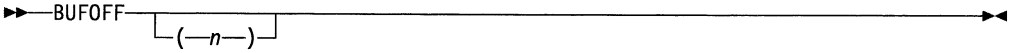
Four control bytes
Data bytes.

The four control bytes contain the length of the record; this value is inserted by data management and requires no action by you. In addition, there can be, at the start of the block, a block prefix field, which can contain the length of the block.

DB-format:
The records are blocked. All other information given for D-format applies to DB-format.

BUFOFF

The BUFOFF option only applies to ASCII data sets. It specifies a *block prefix* field *n* bytes in length at the beginning of each block in an ASCII data set, according to the following syntax:



n is either:

- An integer from 0 to 99
- A variable with attributes FIXED BINARY(31,0) STATIC having an integer value from 0 to 99.

When you are accessing an ASCII data set for input to your program, specifying BUFOFF and *n* identifies to data management how far into the block the beginning of the data is. Specifying BUFOFF without *n* signifies to data management that the first 4 bytes of the data set comprise a block-length field.

When you are creating an ASCII data set for output from your program, PL/I does not allow you to create a prefix field at the beginning of the block using BUFOFF, *unless* it is for data management's use as a 4-byte block-length indicator. In this case, you do not need to specify the BUFOFF option anyway, because for D- or DB-formats PL/I automatically sets up the required field. You *can* code BUFOFF without *n* (though it isn't needed), but that is the only explicit specification of the BUFOFF option that PL/I accepts for output. Therefore, by not coding the BUFOFF option you allow PL/I to set the default values needed for creating your output ASCII data set (4 for D- and DB-formats, 0 for other acceptable formats).

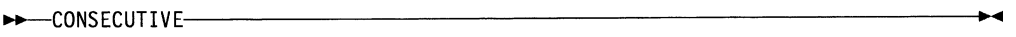
CMDCHN

The CMDCHN option is permitted only with the IBM 3540 Diskette. It allows you to simulate blocked records on the diskette by specifying a CCW chaining factor; this can be 1, 2, 13, or 26. Blocked (FB) records as such are invalid for the 3540, because it is considered to be a unit record device.

If you specify the CMDCHN option, you must explicitly open the file. If you do not specify CMDCHN for a 3540 file, CMDCHN(1) is assumed.

CONSECUTIVE

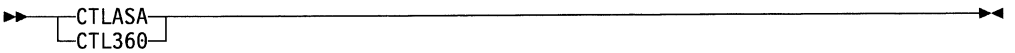
The CONSECUTIVE option defines a file with consecutive data set organization, which is described in this chapter and in "Data set organization" on page 73.



CONSECUTIVE is the default if you don't specify any of the data set organization options (CONSECUTIVE, REGIONAL or VSAM).

CTLASA|CTL360

The print and punch 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.



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.

ASA control characters: Table 13 on page 104 lists the American National Standard control characters for printers and card punches. These control characters cause the specified action to occur before the associated record is printed or punched.

Table 13. American National Standard print and card punch control characters

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

Table 14 lists the American National Standard control characters for the IBM 3525 Card Printer.

Table 14. IBM 3525 Card Printer control characters (CTLASA)

Code	Action
	Space 1 line and print (blank code)
0	Space 2 lines and print
"	Space 3 lines and print
1	Skip to channel 1 and print
2	Skip to channel 2 and print
3	Skip to channel 3 and print
4	Skip to channel 4 and print
5	Skip to channel 5 and print
6	Skip to channel 6 and print
7	Skip to channel 7 and print
8	Skip to channel 8 and print
9	Skip to channel 9 and print
A	Skip to channel 10 and print
B	Skip to channel 11 and print
C	Skip to channel 12 and print

Machine code control characters: The machine code control characters differ according to the type of device. The IBM machine code control characters for printers are listed in Table 15.

Table 15. IBM machine code print control characters

Code byte contents				Action
Print and then act		Act immediately (no printing)		
'00000001'B	'01'X	—		Print only (no space)
'00001001'B	'09'X	'00001011'B	'0B'X	Space 1 line
'00010001'B	'11'X	'00010011'B	'13'X	Space 2 lines
'00011001'B	'19'X	'00011011'B	'1B'X	Space 3 lines
'10001001'B	'89'X	'10001011'B	'8B'X	Skip to channel 1
'10010001'B	'91'X	'10010011'B	'93'X	Skip to channel 2
'10011001'B	'99'X	'10011011'B	'9B'X	Skip to channel 3
'10100001'B	'A1'X	'10100011'B	'A3'X	Skip to channel 4
'10101001'B	'A9'X	'10101011'B	'AB'X	Skip to channel 5
'10110001'B	'B1'X	'10110011'B	'B3'X	Skip to channel 6
'10111001'B	'B9'X	'10111011'B	'BB'X	Skip to channel 7
'11000001'B	'C1'X	'11000011'B	'C3'X	Skip to channel 8
'11001001'B	'C9'X	'11001011'B	'CB'X	Skip to channel 9
'11010001'B	'D1'X	'11010011'B	'D3'X	Skip to channel 10
'11011001'B	'D9'X	'11011011'B	'DB'X	Skip to channel 11
'11100001'B	'E1'X	'11100011'B	'E3'X	Skip to channel 12

Table 16 lists the IBM machine code control characters for stacker selection with the 2540 Card Read Punch.

Table 16. IBM machine code stacker select characters for the 2540 Card Read Punch

Code byte		Action
'00000001'B	'01'X	Select stacker 1
'01000001'B	'41'X	Select stacker 2
'10000001'B	'81'X	Select stacker 3

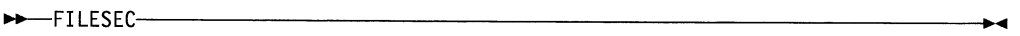
Table 17 on page 106 lists the IBM machine code control characters for printing on cards with the 3525 Card Printer.

Table 17. IBM 3525 Card Printer control characters (CTL360)

Code byte		Action
'00001101'B	'0D'X	Print on line 1
'00010101'B	'15'X	Print on line 2
'00011101'B	'1D'X	Print on line 3
'00100101'B	'25'X	Print on line 4
'00101101'B	'2D'X	Print on line 5
'00110101'B	'35'X	Print on line 6
'00111101'B	'3D'X	Print on line 7
'01000101'B	'45'X	Print on line 8
'01001101'B	'4D'X	Print on line 9
'01010101'B	'55'X	Print on line 10
'01011101'B	'5D'X	Print on line 11
'01100101'B	'65'X	Print on line 12
'01101101'B	'6D'X	Print on line 13
'01110101'B	'75'X	Print on line 14
'01111101'B	'7D'X	Print on line 15
'10000101'B	'85'X	Print on line 16
'10001101'B	'8D'X	Print on line 17
'10010101'B	'95'X	Print on line 18
'10011101'B	'9D'X	Print on line 19
'10100101'B	'A5'X	Print on line 20
'10101101'B	'AD'X	Print on line 21
'10110101'B	'B5'X	Print on line 22
'10111101'B	'BD'X	Print on line 23
'11000101'B	'C5'X	Print on line 24
'11001101'B	'CD'X	Print on line 25

FILESEC

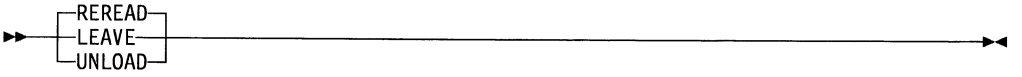
The FILESEC (file security) option is permitted only with IBM 3540 Diskette output files.



The FILESEC option is used to specify that the operator must authorize any future attempts to read the diskette volume.

LEAVE|REREAD|UNLOAD

The magnetic tape handling options LEAVE, REREAD, and UNLOAD allow you to specify the action to be taken when the data set on a magnetic tape volume is opened or closed (or, with a multivolume data set, when volume switching occurs). The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to the load point at which it was mounted. The UNLOAD option rewinds the tape to the load point and unloads the tape.



When either an OPEN or a CLOSE is performed on a magnetic tape file, the current environment of LEAVE, REREAD, or UNLOAD for that file takes effect. If the environment option on the DECLARE statement is left to default to REREAD, then the tape will be rewound as it is opened. Likewise, if no environment option is specified on the CLOSE statement, then the current environment present at OPEN time will take effect. If the environment option is specified on the CLOSE statement

then that option will be used only for the closing of the file; subsequent reopening of the file will use the environment options specified on the DECLARE statement.

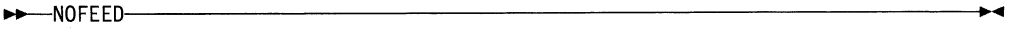
If a data set is first read or written forward and then read backwards in the same program, specify the LEAVE option on either the DECLARE or CLOSE statement of the forward action, and on the DECLARE for the file to be read backwards. This ensures that the tape will not rewind during the close of the forward file or the open of the backwards file. The effects of the options are summarized in Table 18.

Table 18. Effect of tape positioning options

ENVIRONMENT option	Action
LEAVE	Prevents rewind; that is, the tape position is not changed.
REREAD	Rewinds the current volume to the load point. **Note:** This is a different action to the PL/I MVS & VM implementation of REREAD.
UNLOAD	Rewinds and unloads the current volume.

NOFEED

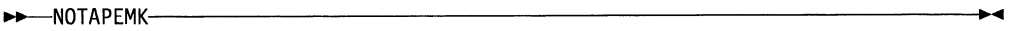
The NOFEED option is permitted only with the IBM 3540 Diskette.



This specifies that the diskette is to remain in position at the end of a job step, so that it can be accessed later without operator intervention.

NOTAPEMK

The NOTAPEMK option only applies to OUTPUT files associated with unlabeled tape data sets (see "Using magnetic tape without standard labels" on page 111).

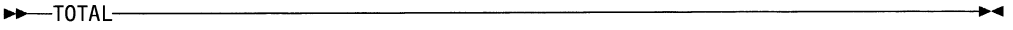


This specifies that a leading tape mark will not be written ahead of the data records on the tape.

TOTAL

In general, run-time library subroutines called from object code perform I/O operations. Under certain conditions, however, the compiler can, when requested, provide in-line code to carry out these operations. This gives faster execution of the I/O statements.

Use the TOTAL option to aid the compiler in the production of efficient object code. In particular, it requests the compiler to use in-line code for certain I/O operations. It specifies that no attributes will be merged from the OPEN statement or the I/O statement or the JCL; if a complete set of attributes can be built up at compile time from explicitly declared and default attributes, then in-line code will be used for certain I/O operations.



The UNDEFINEDFILE condition is raised if any attribute that was not explicitly declared appears on the OPEN statement, or if the I/O statement implies a file attribute that conflicts with a declared or default attribute.

The TOTAL option can only be used when

- the data set organization is CONSECUTIVE
- file attributes are SEQUENTIAL, BUFFERED, and INPUT or OUTPUT
- record format is F, FB, V, VB, or U

The use of in-line I/O code can result in reduced error-handling capability. In particular, if a program-check interrupt or an abend occurs during in-line I/O, the error message produced can contain incorrect offset and statement number information. Also, execution of a GO TO statement in an ERROR ON-unit for such an interrupt can cause a second program check.

Table 19 shows the conditions under which I/O statements are handled in-line.

When in-line code is employed to implement an I/O statement, the compiler gives an informational message.

Table 19. Conditions under which I/O statements are handled in-line (TOTAL option used)

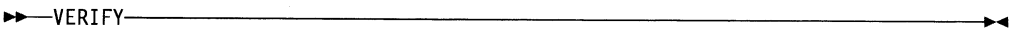
Statement[1]	**Record variable requirements**	**File attribute or ENVIRONMENT option requirements**
READ SET	None	None
READ INTO	Length known at compile time, maximum length for a varying string or area.[2]	RECSIZE known at compile time.[3] SCALARVARYING option if varying string.
WRITE FROM (fixed string)	Length known at compile time.	RECSIZE known at compile time.[3]
WRITE FROM (varying string)		RECSIZE known at compile time.[3] SCALARVARYING option used.
WRITE FROM (area)[2]		RECSIZE known at compile time.[3]
LOCATE	Length known at compile time, maximum length for a varying string or area.[2]	RECSIZE known at compile time.[3] SCALARVARYING if varying string.

Notes:

1. All statements must be found to be valid during compilation. File parameters or file variables are *never* handled by in-line code.
2. Including structures whose last element is an unsubscripted area.
3. You can specify BLKSIZE instead of RECSIZE for unblocked record formats F and U.

VERIFY

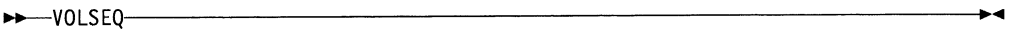
The VERIFY option specifies that a read-check is performed after every write operation.



VERIFY is permitted only for data sets on direct access devices.

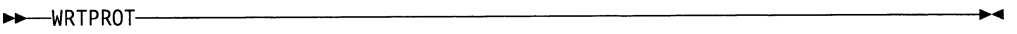
VOLSEQ

The VOLSEQ (volume sequence) option applies only to IBM 3540 Diskette input files. For multivolume data sets, it specifies that sequence checking on the volume serial numbers will be performed.



WRTPROT

The WRTPROT (write protect) option applies only to IBM 3540 Diskette output files. It specifies that the data set that is created will be flagged as a read-only data set.



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 12 on page 100 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 your JCL. The following paragraphs, summarized in Table 20, tell what essential information you must include in the JCL and discuss some of the optional information you can supply.

Table 20. Creating a consecutive data set with record I/O: essential parameters

Storage device	When required	What you must state	Where specified
All	Always	Symbolic device name	File declaration in source program MEDIUM((SYS...)) or EXTENT JCL statement[1] for DASD data sets
		Record format	File declaration (F|FB|V|VB|D|DB|U)
		Record size	File declaration (RECSIZE)
All	Non-standard device assignment	Device assignment	ASSGN statement
Magnetic tape only	Standard labelled tape file	Data set name	TLBL statement
Direct access data sets	Always	Data set name	DLBL statement
		Data set extent	EXTENT statement[2]

Notes:

1. The EXTENT statement overrides the MEDIUM option if both are specified.
2. The EXTENT statement is not always required when using VSE/VSAM Space Management for SAM.

Essential information

When you create a consecutive data set you must specify:

- The symbolic device name of the device that will write your data set (SYSnnn). You can specify this in the MEDIUM ENVIRONMENT option, or for data sets on DASD only, on the EXTENT JCL statement.

- The record format. If you do not specify a record format, the default is F-format.
- The record size.
- The block size if the records are blocked.

If you want your data set stored on a direct-access device, you must also specify:

- The name of the data set (on the DLBL statement).
- The name of the direct access volume that will contain the data set (on the EXTENT statement).
- The physical location of the data set on that volume. This is the starting track number and number of tracks, specified on the EXTENT statement.

For data sets on magnetic tape, you must also specify:

- The name of the data set (on the TLBL statement).
- The ID of the tape volume that will contain the data set. This is specified in the 'file-serial-number' option of the TLBL statement.
- If your data set is not the first (or only) data set on a magnetic-tape volume, you must use the 'volume-sequence-number' option of the TLBL statement to indicate its sequence number on the tape.

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 18 on page 112 for an example of a program that accesses and updates a consecutive data set. If you open the file for output, and the data set already exists on tape or disk, 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 12 on page 100 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.

The operating system does not allow updating a consecutive data set on magnetic tape. To replace or insert records, you must read the data set and write the updated records into a new data set.

You can read a consecutive data set on magnetic tape forward or backward. If you want to read the data set backward, you must give the associated file the BACKWARDS attribute. You can only specify BACKWARDS when the record format is F, FB, or U.

To access a data set, you must identify it to the operating system in your JCL. The following paragraphs describe the essential information you must include in the JCL statements or in your program source, 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 accessing a data set using record-oriented transmission, you must specify:

- The symbolic device name (SYSnnn). You can specify this in the MEDIUM option, or for data sets on direct access devices, on the EXTENT statement.
- The record format, record length, and if records are blocked, the block size. You specify these as ENVIRONMENT options in the file declaration.
- For standard labelled data sets on tape or direct access devices, the data set name. You specify this on the TLBL or DLBL JCL statements.

If the data set follows another data set on a magnetic-tape volume, you must use the 'file-sequence-number' parameter of the TLBL statement to indicate its relative position on the tape.

Using magnetic tape without standard labels

If a magnetic tape data set has nonstandard labels or is unlabeled, you must indicate this by omitting the TLBL statement from your JCL. The ASSGN statement matches the MEDIUM option and points to the device, and the absence of a TLBL statement indicates that the tape on the device does not have standard labels.

PL/I includes no facilities for processing nonstandard labels. These appear to the operating system, and to PL/I, as data sets preceding or following your data set. You can either process the labels as independent data sets or use the MTC JCL statement to bypass them.

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 smaller block size, your data is truncated.

Example of consecutive data sets

Figure 18 on page 112 shows an example of a program that creates and accesses consecutive data sets. The program merges the contents of two data sets and writes them onto a new data set; 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.

Each of the data sets is identified and associated with the appropriate PL/I file by a DLBL statement in the JCL.

```
// JOB    EXAMPLE
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
%PROCESS INT F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 MERGE: PROC OPTIONS(MAIN);
   DCL  INPUT1     FILE RECORD SEQUENTIAL   /* FIRST INPUT FILE     */
                   ENV(FB RECSIZE(15) BLKSIZE(1500)),
        INPUT2     FILE RECORD SEQUENTIAL   /* SECOND INPUT FILE    */
                   ENV(FB RECSIZE(15) BLKSIZE(1500)),
        OUT        FILE RECORD SEQUENTIAL   /* RESULTING MERGED FILE*/
                   ENV(FB RECSIZE(15) BLKSIZE(1500));
   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 18 (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;
/*
// EXEC   LNKEDT
// DLBL   INPUT1,'INPUT.FILE1'
// EXTENT SYS012,SYSWK2
// ASSGN  SYS012,DISK,VOL=VSE111,SHR
// DLBL   INPUT2,'INPUT.FILE2'
// EXTENT SYS013,SYSWK2
// ASSGN  SYS013,DISK,VOL=VSE222,SHR
// DLBL   OUT,'OUTPUT.FILE',0,SD
// EXTENT SYS014,VSE333,1,0,3458,2
// ASSGN  SYS014,DISK,VOL=VSE333,SHR
// EXEC   ,SIZE=128K
/&
```

Figure 18 (Part 2 of 2). Merge sort—creating and accessing a consecutive data set

The program in Figure 19 on page 114 uses record-oriented data transmission to print the table created by the program in Figure 14 on page 96.

```
// JOB    EX#F814
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
%PROCESS INT F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

 PRT: PROC OPTIONS(MAIN);
   DCL TABLE       FILE RECORD INPUT SEQUENTIAL ENV(F RECSIZE(94));
   DCL PRINTER     FILE RECORD OUTPUT SEQUENTIAL
                        ENV(F BLKSIZE(94) CTLASA MEDIUM(SYSLST));
   DCL LINE        CHAR(94);

   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;
/*
// EXEC   LNKEDT
// DLBL   TABLE,'SINE.LIST',0,SD
// EXTENT SYS011,VSE111
// ASSGN  SYS011,DISK,VOL=VSE111,SHR
// EXEC   ,SIZE=128K
/&
```

Figure 19. Printing record-oriented data transmission

Chapter 7. Defining and using regional data sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. It discusses how to create and access regional data sets for the three types of regional organization.

When designing data sets for new applications, it is preferable to use VSAM rather than regional data sets, for the following reasons:

- Regional data sets are PL/I-specific. In general, it is not possible for programs written in other high-level languages to access regional data sets.
- Regional support was implemented in PL/I before VSAM was developed. VSAM provides all of the facilities of regional data sets, and many more options for performance and flexibility.
- VSAM data sets can reside on both FBA and CKD devices, and can be moved between these device types without the need to change application programs. Regional data sets can only reside on CKD devices.
- Direct access of regional data sets can be quicker than that of VSAM 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.

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 in a data transmission statement. With REGIONAL(2) and (3) files, data transmission statements can specify a key as well as the region number, to identify the record to be accessed.

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 organization, in which successive records are written in strict physical sequence; this method does not take full advantage of the characteristics of direct-access storage devices.

You can create a regional data set in a manner similar to a consecutive data set, presenting records in the order of ascending region numbers; alternatively, you can use direct access, in which you present records in random sequence and insert them directly into preformatted regions. Once you create a regional data set, you can access it by using a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. You do not need to specify either a region number or a key if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Records within a regional data set are either actual records containing valid data or dummy records. The nature of the dummy records depends on the type of regional organization; the three types of regional organization are described in this chapter.

Table 21 on page 116 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 UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
SEQUENTIAL UPDATE[3] BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)

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);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression);	EVENT(event-reference)

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 | UNBUFFERED
            [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
            [KEYED]
             UNBUFFERED
             ENVIRONMENT(options);
```

Default file attributes are shown in Table 9 on page 76. The file attributes are described in the *PL/I VSE Language Reference.* Options of the ENVIRONMENT attribute are discussed below.

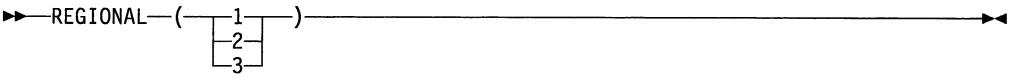
Specifying ENVIRONMENT options

The ENVIRONMENT options applicable to regional data sets are:

```
REGIONAL({1|2|3})
F|V|VS|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
COBOL
COMPAT
BUFFERS(n)
KEYLENGTH(n)
LIMCT(n)
MEDIUM(SYSnnn)
```

REGIONAL option

Use the REGIONAL option to define a file with regional organization.



1 | 2 | 3
specifies REGIONAL(1), REGIONAL(2), or REGIONAL(3), respectively.

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).

REGIONAL(2)
specifies that the data set contains F-format records that have recorded keys. Each region in the data set contains only one record.

REGIONAL(2) differs from REGIONAL(1) in that REGIONAL(2) records contain recorded keys and that records are not necessarily in the specified region; the specified region identifies a starting point.

For data sets created sequentially, the record is written in the specified region.

For files with the DIRECT attribute, a record is written in the first vacant space on or after the track that contains the region number specified in the WRITE statement. For retrieval, the region number specified in the source key is employed to locate the specified region. The method of search is described further in the REGIONAL(2) discussion later in this chapter.

REGIONAL(3)
specifies that the data set contains F-format, V-format, VS-format, or U-format records with recorded keys. Each region in the data set corresponds with a track on a direct-access device and can contain one or more records.

Direct access of a REGIONAL(3) data set employs the region number specified in a source key to locate the required region. Once the region has been located, a sequential search is made for space to add a record, or for a record that has a recorded key identical with that supplied in the source key.

VS-format records can span more than one region. With REGIONAL(3) organization, the use of VS-format removes the limitations on block size imposed by the physical characteristics of the direct-access device. If the record length exceeds the size of a track, or if there is no room left on the current track for the record, the record will be spanned over one or more tracks.

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). REGIONAL(2) and REGIONAL(3) are more appropriate where records are identified by numbers that are thinly distributed over a wide range. You can include in your program an algorithm that derives the region number from the number that identifies a record in such a manner as to optimize the use of space within the data set; duplicate region numbers can occur but, unless they are on the same track, their only effect might be to lengthen the search time for records with duplicate region numbers.

The examples throughout this chapter illustrate typical applications of the three types of regional organization.

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. Recorded keys in a regional data set are never imbedded 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). 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. Imbedded blanks are not allowed in the number; the first imbedded 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 'FF'X 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 'FF'X 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 116 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, 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 data set 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 entire 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.

Example

Creating a REGIONAL(1) data set is illustrated in Figure 20 on page 121. 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.

```
// JOB    EX9
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 CRR1:   PROC OPTIONS(MAIN);
  /*  CREATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY     */

   DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1)
                                              F RECSIZE(20));
   DCL  SYSIN FILE INPUT RECORD ENV(MEDIUM(SYSIPT));
   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;
/*
// EXEC   LNKEDT
// ASSGN  SYS006,3390,VOL=VSE222,SHR
// DLBL   NOS,'PHONE.NUMBERS',,DA
// EXTENT SYS006,VSE222,1,0,3500,19
// EXEC   ,SIZE=128K
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 20. 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 116 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 with 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 6, "Defining and using consecutive data sets" on page 86.

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

Figure 21 on page 123 is an example of a program that updates a REGIONAL(1) 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.

```
// JOB    EX10
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 ACR1: PROC OPTIONS(MAIN);
   /*  UPDATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY      */
    DCL NOS FILE RECORD  KEYED ENV(REGIONAL(1) F RECSIZE(20));
    DCL  SYSIN FILE INPUT RECORD ENV(MEDIUM(SYSIPT));
    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);
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
    END;
   CLOSE FILE(NOS);
   END ACR1;
/*
// EXEC   LNKEDT
// DLBL   NOS,'PHONE.NUMBERS',,DA
// EXTENT SYS006,VSE222
// ASSGN  SYS006,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
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 21. Updating a REGIONAL(1) data set

Using REGIONAL(2) and (3) data sets

In REGIONAL(2) and (3) data sets, each record is identified by a recorded key that immediately precedes the record. The actual position of the record in the data set relative to other records is determined not by its recorded key, but by the region number that you supply in the source key of the WRITE statement that adds the record to the data set.

REGIONAL(2):

The data set is divided into regions in the same manner as REGIONAL(1), and each region can contain one record. The records are fixed length, unblocked.

REGIONAL(3):

Each region number identifies a *track* on the direct-access device that contains the data set. The data set can contain F-, V-, VS-, or U-format records. Each region can contain one or more records, or a segment of a VS-format record.

When you add a record to the data set by direct access, it is written with its recorded key in the first available space after the beginning of the track for the region specified. When a record is read by direct access, the search for a record with the appropriate recorded key begins at the start of the track that contains the region specified. For F- and U-format records, only one track is searched. For V- and VS-format records, the search is extended to five tracks in a manner similar to REGIONAL(2).

A record will not necessarily be located in the physical region specified in the key for that record. When you add a record by direct access, the region number is used as a starting point to search for an available region. The track containing that region and the next *n-1* tracks will be searched (where *n* is the value specified in the LIMCT ENVIRONMENT option, or 5 if LIMCT is not specified), and the record will be placed in the first available space. If the search reaches the end of the data set, it will wrap to the beginning of the data set and continue up to the *n* track total. When you *read* a record by direct access, the same technique is used—if the record with the specified key is not found in the *n* tracks, the KEY condition is raised.

Using keys with REGIONAL(2) and (3) data sets

The character value of the source key can be thought of as having two logical parts—the region number and a comparison key. On output, the comparison key is written as the recorded key; for input, it is compared with the recorded key.

The rightmost 8 characters of the source key make up the region number, which must be the character representation of a fixed decimal integer that does not exceed 16777215 for REGIONAL(2), or 32767 for REGIONAL(3) (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). You can only specify the characters 0 through 9 and the blank character; leading blanks are interpreted as zeros. Imbedded blanks are not allowed in the number; the first imbedded blank, if any, terminates the region number. The comparison key is a character string that occupies the left hand side of the source key, and can overlap or be distinct from the region number, from which it can be separated by other nonsignificant characters.

Specify the length of the comparison key with the KEYLENGTH option of the ENVIRONMENT attribute. If the source key is shorter than the key length you

specify, it is extended on the right with blanks. To retrieve a record, the comparison key must exactly match the recorded key of the record. The comparison key can include the region number, in which case the source key and the comparison key are identical; or, you can use only part of the source key. The length of the comparison key is always equal to KEYLENGTH; if the source key is longer than KEYLENGTH+8, the characters in the source key between the comparison key and the region number are ignored.

When generating the key, you should consider the rules for conversion from arithmetic to character string. For example, the following group is incorrect:

```
DCL KEYS CHAR(8);
DO I=1 TO 10;
   KEYS=I;
   WRITE FILE(F) FROM (R)
     KEYFROM (KEYS);
END;
```

The default for I is FIXED BINARY(15,0), which requires not 8 but 9 characters to contain the character string representation of the arithmetic values. In this example the rightmost digit is truncated.

Consider the following examples of source keys (the character "b" represents a blank):

Example 1:

```
KEY ('JOHNbDOEbbbbbbbbbb3251')
```

The rightmost 8 characters make up the region specification, the relative number of the record. Assume that the file was declared with KEYLENGTH(14). In retrieving a record, the search begins with the beginning of the track that contains the region number 3251, until the record is found having the recorded key of JOHNbDOEbbbbbb.

If the file was declared with KEYLENGTH(22), the search still begins at the same place, but since the comparison and the source key are the same length, the search would be for a record having the recorded key 'JOHNbDOEbbbbbbbbbb3251'.

Example 2:

```
KEY('JOHNbDOEbbbbbbDIVISIONb423bbbbb4627')
```

In this example, the rightmost 8 characters contain leading blanks, which are interpreted as zeros. The search begins at region number 00004627. If KEYLENGTH(14) is specified, the characters DIVISIONb423b will be ignored.

Example 3: Assume that COUNTER is declared FIXED BINARY(21) and NAME is declared CHARACTER(15). You could specify the key as:

```
KEY (NAME || COUNTER)
```

The value of COUNTER will be converted to a character string of 11 characters. (The rules for conversion specify that a binary value of this length, when converted to character, will result in a string of length 11—three blanks followed by eight decimal digits.) The value of the rightmost eight characters of the converted string is taken to be the region specification. Then if the keylength specification is KEYLENGTH(15), the value of NAME is taken to be the comparison specification.

Dummy records

REGIONAL(2) and (3) data sets can contain dummy records. A dummy record is identified by 'FF'X in every byte of the recorded key.

F-format:

The program inserts dummy records either when the data set is created or when a record is deleted. The dummy records are ignored when the program reads the data set.

However, you can replace dummy records with valid data.

V-, VS-, and U-format:

You can identify dummy records because they have dummy recorded keys (all 'FF'X). Records are converted to dummy records only when they are deleted, and you cannot reconvert them to valid records.

Creating REGIONAL(2) and (3) data sets

You can create REGIONAL(2) and (3) data sets either sequentially or by direct access. Table 21 on page 116 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, you must present records in ascending order of region numbers.

REGIONAL(2):

You can only specify one record per region; any error in the sequence, including an attempt to place more than one record in a region, will raise the KEY condition. Any regions you omit from the sequence will be filled with dummy records, and when the file is closed, all remaining space in the data set will be filled with dummy records.

REGIONAL(3):

You can specify the same region number for successive records. For F-format records, any region number you omit from the sequence is filled with dummy records, and the remainder of any unfilled regions is filled with dummy records. If you make an error in the sequence, the KEY condition is raised. If a track becomes filled by records for which the same region number was specified, the region number is incremented by one; an attempt to add a further record with the same region number raises the KEY condition (sequence error).

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 can raise the ERROR condition.

If you use a DIRECT OUTPUT file to create the data set, the whole of the data set is initialized when the file is opened. For F-format records, the space is filled with dummy records, and for V-format, VS-format, and U-format records, the capacity record for each track is written to indicate empty tracks. You can present records in random order, and no condition is raised by duplicate keys or duplicate region specifications. If the data set has F-format records, each record is substituted for the first dummy record found in the search sequence for the region specified on the source key. If the data set has V-format, VS-format, or U-format records, the new record is inserted on the specified track, if sufficient space is available.

Note that for spanned records, space might be required for overflow onto subsequent tracks.

Example

A program for creating REGIONAL data sets is shown in Figure 22 on page 128, and subsequent figures show programs to update sequentially and directly. The programs are coded for REGIONAL(2) data sets, and the differences for REGIONAL(3) are highlighted by shading.

The programs depict a library processing scheme, in which loans of books are recorded and reminders are issued for overdue books. Two data sets, SAMPL.STOCK and SAMPL.LOANS are used. SAMPL.STOCK contains descriptions of the books in the library, and uses the 4-digit book reference numbers as recorded keys; a simple algorithm is used to derive the region numbers from the reference numbers. (It is assumed that there are about 1000 books, each with a number in the range 1000–9999.) SAMPL.LOANS contains records of books that are on loan; each record comprises two dates, the date of issue and the date of the last reminder. Each reader is identified by a 3-digit reference number, which is used to derive the region number in SAMPL.LOANS; the reader and book numbers are concatenated to form the recorded keys.

Figure 22 shows the creation of the data sets SAMPL.STOCK and SAMPL.LOANS. The file LOANS, which is used to create the data set SAMPL.LOANS, is opened for direct output to format the data set; the file is closed immediately without any records being written onto the data set.

For REGIONAL(2), the SAMPL.STOCK data set is created by a DIRECT file because, even if the input data is presented in ascending reference number order, identical region numbers might be derived from successive reference numbers. For REGIONAL(3), SAMPL.STOCK is created sequentially; identical region numbers are acceptable because each region can contain more than one record.

Changes for REGIONAL(3):

1. The LOANS and STOCK files are declared with ENV(REGIONAL(3)).
2. The STOCK file can be opened with the SEQUENTIAL attribute. With REGIONAL(2), if an attempt is made to write more than one record with the same region number to a SEQUENTIAL file, the KEY condition is raised. It is acceptable with REGIONAL(3) because each region can contain more than one record.
3. The REGIONAL(2) data set is divided into 1000 regions, numbered 0-999. The REGIONAL(3) data set has 5 regions, numbered 0-4. For REGIONAL(3), the region number is calculated by:

```
INTER = (NUMBER-1000)/2250;
```

Accessing and updating REGIONAL(2) and (3) data sets

Once you create a REGIONAL(2) or (3) data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. You can only open it for OUTPUT if the entire existing data set is to be deleted and replaced. Table 21 on page 116 shows the statements and options for accessing a regional data set.

```
// JOB    EX14
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
%PROCESS MAR(1,72);
 /*   CREATING A REGIONAL(2) OR (3) DATA SET - LIBRARY LOANS        */
CRREG: PROC OPTIONS(MAIN);

DCL LOANS FILE RECORD KEYED ENV(REGIONAL(2)     1
                                F BLKSIZE(12) KEYLENGTH(7));
DCL STOCK FILE RECORD KEYED ENV(REGIONAL(2)
                                F BLKSIZE(77) KEYLENGTH(4));

DCL SYSIN FILE STREAM INPUT ENV(MEDIUM(SYSIPT));

DCL 1   BOOK,
     2   AUTHOR CHAR(25),
     2   TITLE  CHAR(50),
     2   QTY    FIXED DEC(3);

DCL NUMBER CHAR(4);
DCL INTER FIXED DEC(5);
DCL REGION CHAR(8);
DCL EOF BIT(1) INIT('0'B);

      /* INITIALIZE (FORMAT) LOANS DATA SET                          */

      OPEN FILE(LOANS) DIRECT OUTPUT;
      CLOSE FILE(LOANS);

      ON ENDFILE(SYSIN) EOF='1'B;
      OPEN FILE(SYSIN);
      OPEN FILE(STOCK) DIRECT OUTPUT;          2

      GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
      DO WHILE (¬EOF);
         INTER = (NUMBER-1000)/9;              3
         REGION = INTER;
         WRITE FILE(STOCK) FROM(BOOK) KEYFROM(NUMBER||REGION);
         PUT FILE(SYSPRINT) SKIP EDIT (BOOK) (A);
         GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
      END;

      CLOSE FILE(STOCK);
   END CRREG;
/*
// EXEC   LNKEDT
// DLBL   LOANS,'SAMPL.LOANS',0,DA
// EXTENT SYS005,VSE222,1,0,3500,5
// DLBL   STOCK,'SAMPL.STOCK',0,DA
// EXTENT SYS005,VSE222,1,0,3515,25
// ASSGN  SYS005,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
'1015' 'W.SHAKESPEARE' 'MUCH ADO ABOUT NOTHING'  1
'1214' 'L.CARROLL' 'THE HUNTING OF THE SNARK'  1
'3079' 'G.FLAUBERT' 'MADAME BOVARY'  1
'3083' 'V.M.HUGO' 'LES MISERABLES'  2
'3085' 'J.K.JEROME' 'THREE MEN IN A BOAT'  2
'4295' 'W.LANGLAND' 'THE BOOK CONCERNING PIERS THE PLOWMAN'  1
'5999' 'O.KHAYYAM' 'THE RUBAIYAT OF OMAR KHAYYAM'  3
'6591' 'F.RABELAIS' 'THE HEROIC DEEDS OF GARGANTUA AND PANTAGRUEL'  1
'8362' 'H.D.THOREAU' 'WALDEN, OR LIFE IN THE WOODS'  1
'9795' 'H.G.WELLS' 'THE TIME MACHINE'  3
/*
/&
```

Figure 22. Creating a REGIONAL(3) data set

Sequential access

To open a SEQUENTIAL file that is used to access a REGIONAL(2) or (3) data set, use either the INPUT or UPDATE attribute. You must not include the KEY option in the data transmission statements, but the file can have the KEYED attribute since you can use the KEYTO option.

With the KEYTO option you can specify that the *recorded key only* is to be assigned to the specified variable. If the character string referenced in the KEYTO option has more characters than you specify in the KEYLENGTH option, the value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than you specify in KEYLENGTH, the value returned is truncated on the right.

Sequential access is in the physical order of the records as they exist on the data set. Records are retrieved in this order, and not necessarily in the order in which they were added to the data set. The recorded keys do not affect the order of sequential access. Dummy records are not retrieved.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(2) or (3) data set are identical with those for a CONSECUTIVE data set.

Direct access

To open a DIRECT file that is used to process a REGIONAL(2) or (3) data set, use either the INPUT or the UPDATE attribute. You must include source keys in all data transmission statements; the DIRECT attribute implies the KEYED attribute.

Using direct input, you can read any record by supplying its region number and its recorded key; in direct update, you can read or delete existing records or add new ones.

Retrieval Dummy records are not made available by a READ statement. The KEY condition is raised if a record with the specified recorded key is not found.

Addition In a REGIONAL(2) data set, a WRITE statement substitutes the new record for a dummy record in the region specified by the source key, or in the next available dummy record if the specified region is already occupied.

In a REGIONAL(3) data set with F-format records, a WRITE statement substitutes the new record for a dummy record in the specified region. If the data set has V-format, VS-format, or U-format records, a WRITE statement inserts the new record after any records already present in the region if space is available.

Deletion A record you specify by the source key in a DELETE statement is converted to a dummy record. You can reuse the space formerly occupied by an F-format record; space formerly occupied by V-format, VS-format, or U-format records is not available for reuse.

Replacement The record you specify by the source key in a REWRITE statement must exist; you cannot use a REWRITE statement to replace a dummy record. When a VS-format record is replaced, the new one must not be shorter than the old.

Note: If a track contains records with duplicate recorded keys, the record farthest from the beginning of the track will never be retrieved during direct access.

Example

The data set SAMPL.LOANS, described in "Example" on page 127, is updated directly in Figure 23 on page 131. Each item of input data, read from a source input, comprises a book number, a reader number, and a code to indicate whether it refers to a new issue (I), a returned book (R), or a renewal (A). The date is written in both the issue-date and reminder-date portions of a new record or an updated record.

A sequential update of the same data set is shown in the program in Figure 24 on page 132. The sequential update file (LOANS) processes the records in the data set SAMPL.LOANS, and a direct input file (STOCK) obtains the book description from the data set SAMPL.STOCK for use in a reminder note. Each record from SAMPL.LOANS is tested to see whether the last reminder was issued more than a month ago; if necessary, a reminder note is issued and the current date is written in the reminder-date field of the record.

Again, the programs are coded for REGIONAL(2) data sets, and the changes needed for REGIONAL(3) are highlighted by shading.

Changes for REGIONAL(3): In Figure 23 on page 131:

1. The LOANS file is declared with ENV(REGIONAL(3)).
2. For the REGIONAL(2) data set, the region number is the same as the borrower's 3-digit reference number. The REGIONAL(3) data set is divided into 3 regions, numbered 0-2. For REGIONAL(3), the region number is calculated by:

```
SELECT;
   WHEN(READER < '034') REGION = '00000000';
   WHEN(READER < '067') REGION = '00000001';
   OTHERWISE            REGION = '00000002';
END;
```

In Figure 24 on page 132:

1. The LOANS and STOCK files are declared with ENV(REGIONAL(3)).
2. For REGIONAL(3), the region number is calculated by:

```
INTER = (BKNO-1000)/2250;
```

```
// JOB    EX15
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
%PROCESS MAR(1,72);
 DUREG: PROC OPTIONS(MAIN);
 /* UPDATING A REGIONAL(2) OR (3) DATA SET DIRECTLY - LIBRARY LOANS*/

   DCL LOANS FILE RECORD UPDATE DIRECT KEYED     1
             ENV(REGIONAL(2) F BLKSIZE(12) KEYLENGTH(7));
   DCL 1   RECORD,
        2   (ISSUE,REMINDER) CHAR(6);

   DCL SYSIN FILE RECORD INPUT SEQUENTIAL ENV(MEDIUM(SYSIPT));
   DCL SYSIN_REC BIT(1) INIT('1'B);
   DCL 1   CARD,
        2   BOOK CHAR(4),
        2   CARD_1 CHAR(5),
        2   READER CHAR(3),
        2   CARD_2 CHAR(7),
        2   CODE   CHAR(1),
        2   CARD_3 CHAR(1),
        2   DATE   CHAR(6),
        2   CARD_4 CHAR(53);

   DCL REGION CHAR(8) INIT((8)'0');

   ON ENDFILE(SYSIN) SYSIN_REC= '0'B;
   OPEN FILE(SYSIN),FILE(LOANS);
   READ FILE(SYSIN) INTO(CARD);

   DO WHILE(SYSIN_REC);
      ISSUE,REMINDER = CARD.DATE;
      SUBSTR(REGION,6,3) = CARD.READER;          2
      SELECT(CODE);
         WHEN('I') WRITE FILE(LOANS) FROM(RECORD)   /* New issue */
                   KEYFROM(READER||BOOK||REGION);
         WHEN('R') DELETE FILE(LOANS)               /* Returned  */
                   KEY    (READER||BOOK||REGION);
         WHEN('A') REWRITE FILE(LOANS) FROM(RECORD) /* Renewal   */
                   KEY    (READER||BOOK||REGION);
         OTHERWISE PUT FILE(SYSPRINT) SKIP LIST  /* Invalid code */
                   ('INVALID CODE: ',BOOK,READER);
      END;
      PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
      READ FILE(SYSIN) INTO(CARD);
   END;
   CLOSE FILE(SYSIN),FILE(LOANS);
 END DUREG;
/*
// EXEC   LNKEDT
// DLBL   LOANS,'SAMPL.LOANS',,DA
// EXTENT SYS005,VSE222
// ASSGN  SYS005,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
5999     003       I 940921
3083     091       I 940927
1214     049       I 941001
5999     003       A 941005
3083     091       R 941005
3517     095       X 941007
/*
/&
```

Figure 23. Updating a REGIONAL(3) data set directly

```
// JOB    EX16
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
%PROCESS MAR(1,72);
 SUREG: PROC OPTIONS(MAIN);
 /* UPDATING A REGIONAL(2) OR (3) DATA SET SEQUENTIALLY - LIB LOANS   */

   DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED     1
             ENV(REGIONAL(2) F BLKSIZE(12) KEYLENGTH(7));

   DCL LOANS_REC BIT(1) INIT('1'B);
   DCL 1   RECORD,
        2   (ISSUE,REMINDER) CHAR(6);
   DCL LOANKEY CHAR(7),
       READER  CHAR(3) DEF LOANKEY,
       BKNO    CHAR(4) DEF LOANKEY POS(4);
   DCL STOCK FILE RECORD DIRECT INPUT KEYED          1
             ENV(REGIONAL(2) F BLKSIZE(77) KEYLENGTH(4));
   DCL 1   BOOK,
        2   AUTHOR CHAR(25),
        2   TITLE  CHAR(50),
        2   QTY    FIXED DEC(3);
   DCL TODAY CHAR(6);  /*YYMMDD*/
   DCL INTER FIXED DEC(5),
       REGION CHAR(8);

   TODAY = '941003';
   OPEN FILE (LOANS), FILE(STOCK);
   ON ENDFILE(LOANS) LOANS_REC = '0'B;
   READ FILE(LOANS) INTO(RECORD) KEYTO(LOANKEY);
   X = 1;

   DO WHILE(LOANS_REC);
      PUT FILE(SYSPRINT) SKIP EDIT
      (X,'REM DATE ',REMINDER,' TODAY ',TODAY) (A(3),A(9),A,A(7),A);
      X = X+1;

      IF REMINDER < TODAY THEN
         DO;
            INTER = (BKNO-1000)/9;      2
            REGION = INTER;
            READ FILE(STOCK) INTO(BOOK) KEY(BKNO||REGION);
            REMINDER = TODAY;
            PUT FILE(SYSPRINT) SKIP EDIT
              ('NEW REM DATE',REMINDER,READER,AUTHOR,TITLE)
              (A(12),A,X(2),A,X(2),A,X(2),A);
            REWRITE FILE(LOANS) FROM(RECORD);
         END;
      READ FILE(LOANS) INTO(RECORD) KEYTO(LOANKEY);
   END;
   CLOSE FILE(LOANS),FILE(STOCK);
 END SUREG;
/*
// EXEC   LNKEDT
// DLBL   LOANS,'SAMPL.LOANS',,DA
// EXTENT SYS005,VSE222
// DLBL   STOCK,'SAMPL.STOCK',,DA
// EXTENT SYS005,VSE222
// ASSGN  SYS005,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
/&
```

Figure 24. Updating a REGIONAL(3) data set sequentially

Essential information for creating and accessing regional data sets

To create a regional data set, you must give the operating system certain information, both in your PL/I program and in the JCL statements that define 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:

- The symbolic device name of the device that will write your data set (SYSnnn). You can specify this in the MEDIUM ENVIRONMENT option or on the EXTENT JCL statement.
- The record format and record size.
- The type of organization, either REGIONAL(1), REGIONAL(2), or REGIONAL(3).
- For REGIONAL(2) and (3), you must also state the length of the recorded key in the KEYLENGTH option. See "Using keys with REGIONAL(2) and (3) data sets" on page 124 for a description of how the recorded key is derived from the source key supplied in the KEYFROM option.
- The volume serial number of the DASD volume that will contain the data set. You specify this in the EXTENT JCL statement (the 'serial-number' parameter).

In the DLBL statement, you must always specify the data set organization as direct by coding DA in the 'codes' parameter.

REGIONAL(3) compatibility with DOS PL/I format

REGIONAL(3) F-format data sets created by PL/I VSE are incompatible with those created by DOS PL/I, and vice versa. When creating these data sets, DOS PL/I clears disk tracks, then adds new records in the unused space of each track. PL/I VSE, however, pre-formats disk tracks with dummy records, then adds new records by replacing the dummy records.

If you recompile all your DOS PL/I applications with PL/I VSE, then we recommend that you also migrate your DOS PL/I-created REGIONAL(3) F-format data sets to the PL/I VSE format (for details, see the *PL/I VSE Migration Guide).*

If a REGIONAL(3) F-format data set must be used by both DOS PL/I and PL/I VSE applications, then declare its associated files in PL/I VSE using the ENVIRONMENT(COMPAT) option. This option forces PL/I VSE to create and use REGIONAL(3) F-format data sets in the same way as DOS PL/I. Note that this restricts the search for records and free space to one track (same as DOS PL/I).

If ENVIRONMENT(COMPAT) is coded, then ENVIRONMENT(LIMCT) is ignored.

Chapter 8. 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 four types of VSAM data sets that PL/I supports—entry-sequenced, key-sequenced, relative-record, and variable-length 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

To execute a program that accesses a VSAM data set, you need to relate the data set name to your program's file. This is done in your JCL by means of a DLBL statement. See "Associating data sets with files" on page 66 for information on coding your DLBL statement.

The 'codes' field in the DLBL statement should always be set to VSAM, for example:

```
// DLBL PL1FILE,'VSAMDS',,VSAM
```

Pairing an alternate index path with a file

When using an alternate index, you simply specify the name of the *path* as the data set name in the DLBL statement associating the base data set/alternate index pair with your PL/I file. Before using an alternate index, you should be aware of the restrictions on processing; these are summarized in Table 23 on page 142.

Given a PL/I file called PL1FILE and the alternate index path called PERSALPH, the JCL statement required would be:

```
// DLBL PL1FILE,'PERSALPH',,VSAM
```

VSAM organization

PL/I provides support for four types of VSAM data sets:

- Key-sequenced data sets (KSDS)
- Entry-sequenced data sets (ESDS)
- Relative-record data sets (RRDS)
- Variable-length relative-record data sets (VRDS)

They are all ordered, and they can all have keys associated with their records. Both sequential and keyed access are possible with all four types.

Although only KSDS have keys as part of their logical records, keyed access is also possible for ERDS (using relative-byte addresses), RRDS, and VRDS (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 RRDS, 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.

There are two types of index for VSAM data sets:

Prime index (KSDS only). Each KSDS has a prime index, created when the data set is defined.

Alternate index Each KSDS or ESDS may have one or more alternate indexes. Defining an alternate index for an ESDS enables you to treat the ESDS, in general, as a KSDS. An alternate index on a KSDS enables a field in the logical record different from that in the prime index to be used as the key field.

Alternate indexes can be either *nonunique*, in which duplicate keys are allowed, or *unique*, in which they are not. The prime index can never have duplicate keys.

Any change in a data set that has alternate indexes must be reflected in all the indexes if they are to remain useful. This activity is known as *index upgrade*, and is done by VSAM for any index in the *index upgrade set* of the data set. (For a KSDS, the prime index is always a member of the index upgrade set.) However, you must avoid making changes in the data set that would cause duplicate keys in the prime index or in a unique alternate index.

Before using a VSAM data set for the first time, you need to define it to the system with the DEFINE command of Access Method Services, which you can use to completely define the type, structure, and required space of the data set. This command also defines the data set's indexes (together with their key lengths and locations) and the index upgrade set if the data set has one or more alternate indexes. A VSAM data set is thus “created” by Access Method Services.

The operation of writing the initial data into a newly-created VSAM data set is referred to as *loading* in this publication.

Use the three different types of data sets according to the following purposes:

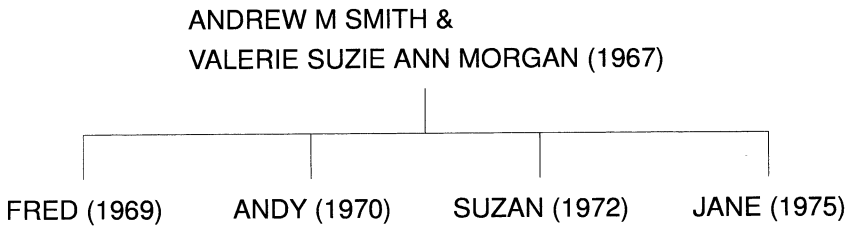
- Use *entry-sequenced data sets* for data that you primarily access in the order in which it was created (or the reverse order).
- Use *key-sequenced data sets* when you normally access records through keys within the records (for example, a stock-control file where the part number is used to access a record).

- Use *relative-record data sets* for data in which each item has a particular number, and you normally access the relevant record by that number (for example, a telephone system with a record associated with each number).

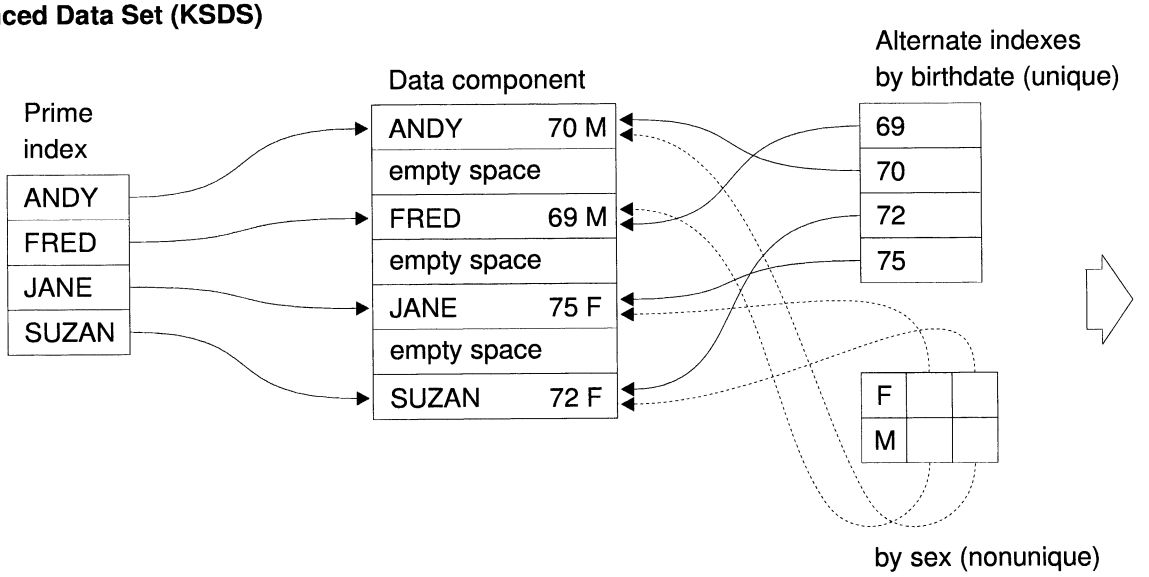
You can access records in all types of VSAM data sets either directly by means of a key, or sequentially (backward or forward). You can also use a combination of the two ways: Select a starting point with a key and then read forward or backward from that point.

You can create alternate indexes for key-sequenced and entry-sequenced data sets. You can then access your data in many sequences or by one of many keys. For example, you could take a data set held or indexed in order of employee number and index it by name in an alternate index. Then you could access it in alphabetic order, in reverse alphabetic order, or directly using the name as a key. You could also access it in the same kind of combinations by employee number.

Figure 25 on page 137 and Table 22 on page 138 show how the same data could be held in the four different types of VSAM data sets and illustrates their respective advantages and disadvantages.



Key-Sequenced Data Set (KSDS)



Entry-Sequenced Data Set (ESDS)

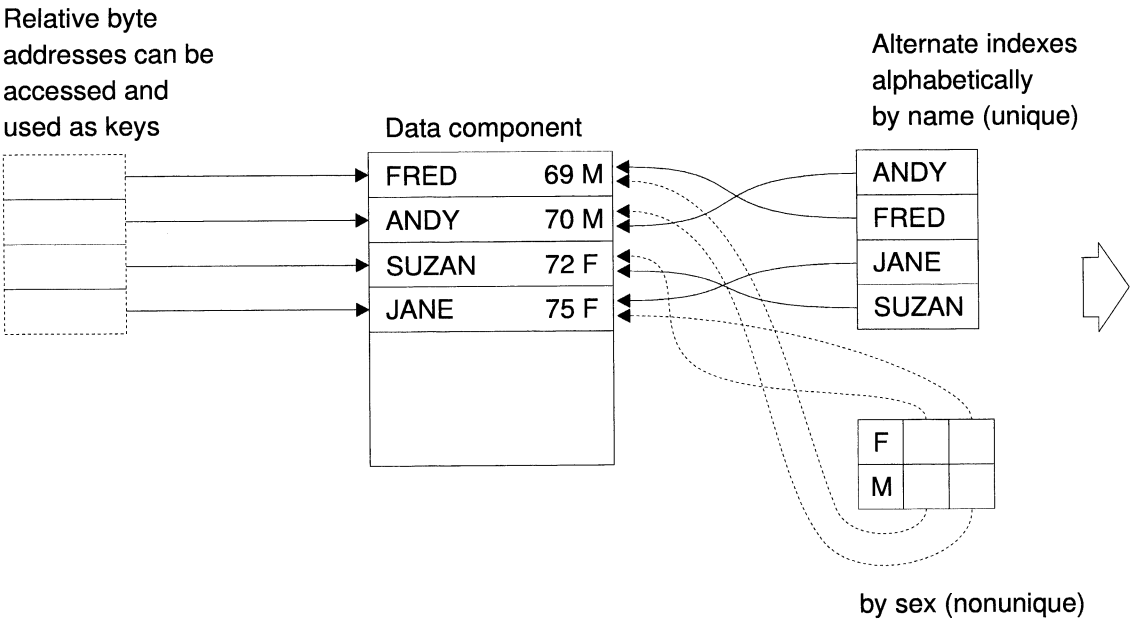
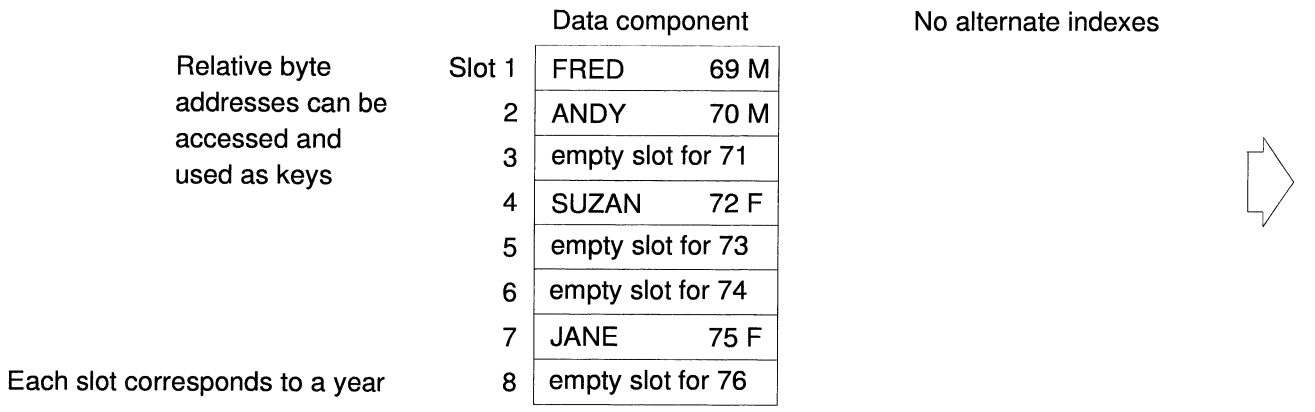


Figure 25 (Part 1 of 2). Information storage in VSAM data sets of different types

Relative-Record Data Set (RRDS)



Variable-length Relative-Record Data Set (VRDS)

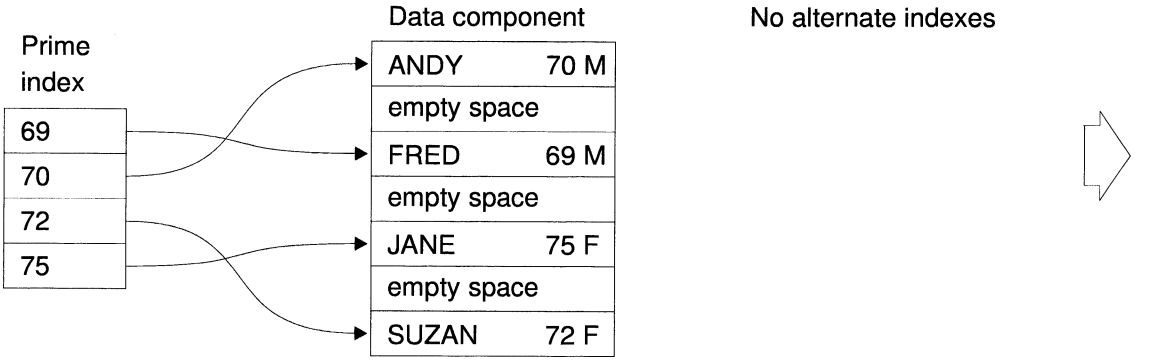


Figure 25 (Part 2 of 2). Information storage in VSAM data sets of different types

Table 22 (Page 1 of 2). 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 or unique alternate 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

Table 22 (Page 2 of 2). Types and advantages of VSAM data sets

Data set type	Method of loading	Method of reading	Method of updating	Pros and cons
Entry-Sequenced	Sequentially (forward only) The RBA of each record can be obtained and used as a key	SEQUENTIAL backward or forward KEYED using unique alternate index or RBA Positioning by key followed by sequential either backward or forward	New records at end only Existing records cannot have length changed Access may be sequential or KEYED using alternate index 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 No alternate index Fixed length records ***Uses*** For use where records will be accessed by number
Variable-length Relative-Record	Sequentially starting from record 1 KEYED specifying number of record 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 Insertion of new records allowed Record replacement allowed with records of the same or different lengths	***Advantages*** Compared to RRDS: • Structure not tied to numbering sequences • Variable-length records • More efficient storage for sparsely-populated data sets ***Disadvantages*** No alternate index ***Uses*** For use where records are variable-length, and will be accessed by number

Keys for VSAM data sets

All VSAM data sets can have keys associated with their records. For key-sequenced data sets, and for entry-sequenced data sets accessed via an alternate index, the key is a defined field within the logical record. For entry-sequenced data sets, the key is the *relative byte address* (RBA) of the record. For RRDS and VRDS, the key is a *relative record number*.

Keys for indexed VSAM data sets

Keys for key-sequenced data sets and for entry-sequenced data sets accessed via an alternate index are part of the logical records recorded on the data set. You define the length and location of the keys when you create the data set.

The ways you can reference the keys in the KEY, KEYFROM, and KEYTO options are as described under "KEY(expression) Option," "KEYFROM(expression) Option," and "KEYTO(reference) Option" in the *PL/I VSE Language Reference* book.

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 and VRDS 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 four types of VSAM data sets and two 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.

Figure 25 on page 137 shows you the possibilities available with the types of VSAM data sets. 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 or VRDS.
2. Determine how you will load the data set. Note that you must load a KSDS in primary key sequence; thus an ESDS with an alternate index path can be a more practical alternative for some applications.
3. Determine whether you require access through an alternate index path. These are only supported on KSDS and ESDS. If you require an alternate index path, determine whether the alternate index will have unique or nonunique keys. Use of nonunique keys can limit key processing. However, it might also be impractical to assume that you will use unique keys for all future records; if you attempt to insert a record with a nonunique key in an index that you have created for unique keys, it will cause an error.
4. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported. Figure 26 on page 142 and Table 23 on page 142 might be helpful.

Table 24 on page 152, Table 25 on page 156, and Table 27 on page 170 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	VRDS
	VRDS	VRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	
OUTPUT	ESDS	ESDS	KSDS
	RRDS	KSDS	RRDS
	VRDS	RRDS	VRDS
		VRDS	Path(U)
UPDATE	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	VRDS
	VRDS	VRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	

Key: ESDS Entry-sequenced data set
KSDS Key-sequenced data set
RRDS Relative-record data set
VRDS Variable-length relative-record data set
Path(N) Alternate index path with nonunique keys
Path(U) Alternate index path with unique keys

You can combine the attributes on the left with those at the top of the figure for the data sets and paths shown. For example, only an ESDS, RRDS, and VRDS can be SEQUENTIAL OUTPUT.

Figure 26. VSAM data sets and allowed file attributes

Table 23. Processing allowed on alternate index paths

Base cluster type	Alternate index key type	Processing	Restrictions
KSDS	Unique key	As normal KSDS	Cannot modify key of access.
	Nonunique key	Limited keyed access	Cannot modify key of access.
ESDS	Unique key	As KSDS	No deletion. Cannot modify key of access.
	Nonunique key	Limited keyed access	No deletion. Cannot modify key of access.

Defining files for VSAM data sets

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

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

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             UNBUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

Table 9 on page 76 shows the default attributes. The file attributes are described in the *PL/I VSE 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 26 on page 142 shows the compatible combinations.

Specifying ENVIRONMENT options

Many of the options of the ENVIRONMENT attribute affecting data set structure are not needed for VSAM data sets. If you specify them, they are either ignored or are used for checking purposes. If those that are checked conflict with the values defined for the data set, the UNDEFINEDFILE condition is raised when an attempt is made to open the file.

The ENVIRONMENT options applicable to VSAM data sets are:

```
BKWD
BUFND(n)
BUFNI(n)
BUFSP(n)
DSN(n)
COBOL
GENKEY
PASSWORD(password-specification)
REUSE
SCALARVARYING
SKIP
VSAM
```

The COBOL 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 9 on page 76 shows which options are ignored for VSAM. Table 9 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.

Sequential reads (that is, reads without the KEY option) retrieve the previous record in sequence. For indexed data sets, the previous record is, in general, the record with the next lower key. However, if you are accessing the data set via a nonunique alternate index, records with the same key are recovered in their normal sequence. For example, if the records are:

```
A B C1 C2 C3 D E
```

where C1, C2, and C3 have the same key, they are recovered in the sequence:

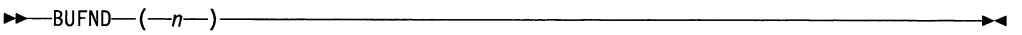
```
E D C1 C2 C3 B A
```

When a file with the BKWD option is opened, the data set is positioned at the last record. ENDFILE is raised in the normal way when the start of the data set is reached.

Do not specify the BKWD option with either the REUSE option or the GENKEY option. Also, the WRITE statement is not allowed for files declared with the BKWD option.

BUFND option

Use the BUFND option to specify the number of data buffers required for a VSAM data set.

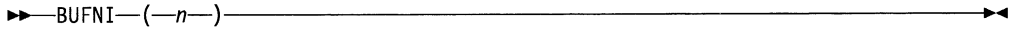


n specifies an integer, or a variable with the attributes FIXED BINARY(31) STATIC.

Multiple data buffers help performance when the file has the SEQUENTIAL attribute and you are processing long groups of contiguous records sequentially.

BUFNI option

Use the BUFNI option to specify the number of index buffers required for a VSAM key-sequenced data set.

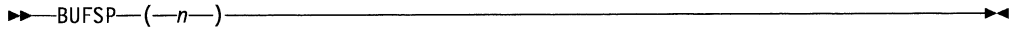


n specifies an integer, or a variable with the attributes FIXED BINARY(31) STATIC.

Multiple index buffers help performance when the file has the KEYED attribute. Specify at least as many index buffers as there are levels in the index.

BUFSP option

Use the BUFSP option to specify, in bytes, the total buffer space required for a VSAM data set (for both the data and index components).

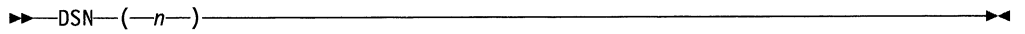


n specifies an integer, or a variable with the attributes FIXED BINARY(31) STATIC.

It is usually preferable to specify the BUFNI and BUFND options rather than BUFSP.

DSN option

The DSN (data set name sharing) option allows you to specify the number of strings to be allocated to the data set for shared access.

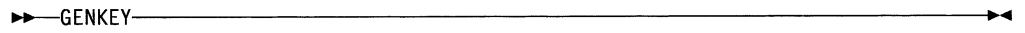


n specifies an integer, or a variable with the attributes FIXED BINARY(31) STATIC. The first opened file with a DSN option determines the number of strings allocated to that data set. VSAM uses *n* to set the ACB BSTRNO and STRNO values. Subsequent opened files for the same data set require a DSN option to participate in DSN sharing, but *n* is ignored.

All files opened with a DSN option use ACB MACRF=OUT. VSE VSAM does not allow mixed MACRF=INIOUT ACBs for DSN sharing. This does not affect processing of the PL/I INPUT or UPDATE options.

GENKEY option

The GENKEY (generic key) option enables you to classify keys recorded in a data set into groups based on the partial contents of the keys, and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.



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. 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.

In the following example, a key length of more than 3 bytes is assumed:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (VSAM GENKEY);
          .
          .
          .
       READ FILE(IND) INTO(INFIELD)
                      KEY ('ABC');
          .
          .
          .
NEXT:  READ FILE (IND) INTO (INFIELD);
          .
          .
          .
       GO TO NEXT;
```

The first READ statement causes the first record in the data set whose key begins with "ABC" to be read into INFIELD; each time the second READ statement is executed, the 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".

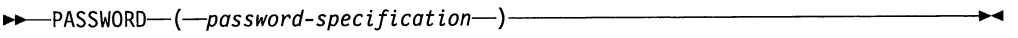
If the data set contains no records with keys in the specified class, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

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 for the data set. Key length is an attribute of the data set, and is specified when the data set is created using the Access Method Services utility (the KEYS parameter of the DEFINE CLUSTER command). 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 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).

PASSWORD option

When you define a VSAM data set to the system (using the DEFINE command of Access Method Services), you can associate READ and UPDATE passwords with it. From that point on, you must include the appropriate password in the declaration of any PL/I file that you use to access the data set.



password-specification
is a character constant or character variable that specifies the password for the type of access your program requires. If you specify a constant, it must not contain a repetition factor; if you specify a variable, it must be level-1, element, static, and unsubscripted.

The character string is padded or truncated to 8 characters and passed to VSAM for inspection. If the password is incorrect, the system operator is given a number of chances to specify the correct password. You specify the number of chances to be allowed when you define the data set. After this number of unsuccessful tries, the UNDEFINEDFILE condition is raised.

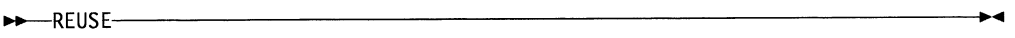
The three levels of password supported by PL/I are:

- Master
- Update
- Read.

Specify the highest level of password needed for the type of access that your program performs.

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.



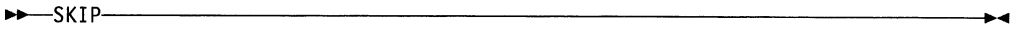
The data set is treated as an empty data set each time the file is opened. Any secondary allocations for the data set are released, and the data set is treated exactly as if it were being opened for the first time.

Do not associate a file that has the REUSE option with a data set that has alternate indexes or the BKWD option, and do not open it for INPUT or UPDATE.

The REUSE option takes effect only if you specify REUSE in the Access Method Services DEFINE CLUSTER command.

SKIP option

Use the SKIP option of the ENVIRONMENT attribute to specify that the VSAM OPTCD "SKP" is to be used wherever possible. It is applicable to key-sequenced data sets that you access by means of a KEYED SEQUENTIAL INPUT or UPDATE file.



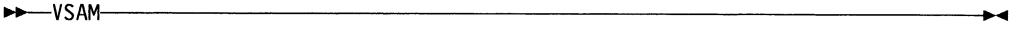
You should specify this option for the file if your program accesses individual records scattered throughout the data set, but does so primarily in ascending key order.

Omit this option if your program reads large numbers of records sequentially without the use of the KEY option, or if it inserts large numbers of records at specific points in the data set (mass sequential insert).

It is never an error to specify (or omit) the SKIP option; its effect on performance is significant only in the circumstances described.

VSAM option

Specify the VSAM option for VSAM data sets.



Performance options

SKIP, BUFND, BUFNI, and BUFSP are options you can specify to optimize VSAM's performance. You can also specify the buffer options on the DLBL statement of your JCL; they are explained in the *VSE/ESA System Control Statements* manual.

Defining files for alternate index paths

VSAM allows you to define alternate indexes on key sequenced and entry sequenced data sets. This enables you to access key sequenced data sets in a number of ways other than from the prime index. This also allows you to index and access entry sequenced data sets by key or sequentially in order of the keys. Consequently, data created in one form can be accessed in a large number of different ways. For example, an employee file might be indexed by personnel number, by name, and also by department number.

When an alternate index has been built, you actually access the data set through a third object known as an alternate index *path* that acts as a connection between the alternate index and the data set.

Two types of alternate indexes are allowed—unique key and nonunique key. For a unique key alternate index, each record must have a different alternate key. For a nonunique key alternate index, any number of records can have the same alternate key. In the example suggested above, the alternate index using the names could be a unique key alternate index (provided each person had a different name). The alternate index using the department number would be a nonunique key alternate index because more than one person would be in each department. An example of alternate indexes applied to a family tree is given in Figure 25 on page 137.

In most respects, you can treat a data set accessed through a unique key alternate index path like a KSDS accessed through its prime index. You can access the records by key or sequentially, you can update records, and you can add new records. If the data set is a KSDS, you can delete records, and alter the length of updated records. Restrictions and allowed processing are shown in Table 23 on page 142. When you add or delete records, all indexes associated with the data set are by default altered to reflect the new situation.

In data sets accessed through a nonunique key alternate index path, the record accessed is determined by the key and the sequence. The key can be used to establish positioning so that sequential access can follow. The use of the key accesses the first record with that key. When the data set is read backwards, only the order of the keys is reversed. The order of the records with the same key remains the same whichever way the data set is read.

Using files defined for non-VSAM data sets

In most cases, if your PL/I program uses files declared with ENVIRONMENT (CONSECUTIVE) or ENVIRONMENT(INDEXED) or with no ENVIRONMENT, it can access VSAM data sets without alteration. If your program uses REGIONAL files, you must alter it and recompile before it can use VSAM data sets. PL/I can detect that a VSAM data set is being opened and can provide the correct relationship between the data set and the file.

The aspects of compatibility that affect your usage of VSAM if your data sets or programs were created for other access methods are as follows:

- Your data sets will need to be re-created as VSAM data sets. The Access Method Services REPRO command re-creates data sets in VSAM format. This command is described in the *VSE/VSAM Commands and Macros* manual.
- All VSAM key-sequenced data sets have imbedded keys, even if they have been converted from ISAM data sets with nonimbedded keys.
- The JCL DLBL statements that refer to the data sets will need to be changed.
- You need to determine whether or not your programs needs to be altered to allow them to use VSAM data sets. A brief discussion of this is given later in this section.

CONSECUTIVE files

For CONSECUTIVE files, compatibility depends on the ability of the PL/I routines to recognize the data set type and use the correct access method.

You should realize, however, that there is no concept of fixed-length records in VSAM. Therefore, if your program relies on the RECORD condition to detect incorrect length records, it will not function in the same way using VSAM data sets as it does with non-VSAM data sets.

INDEXED files

Complete compatibility is provided for INDEXED files. For files that you declare with the INDEXED ENVIRONMENT option, the PL/I library routines recognize a VSAM data set and will process it as VSAM.

However, because ISAM record handling differs in detail from VSAM record handling, use of VSAM processing might not always give the required result.

Adapting existing programs for VSAM

You can readily adapt existing programs with indexed, consecutive, or REGIONAL(1) files for use with VSAM data sets. As indicated above, programs with consecutive or indexed files might not need alteration. Programs with REGIONAL(1) data sets require only minor revision, but programs with REGIONAL(2) or (3) files need restructuring before you can use them with VSAM data sets.

CONSECUTIVE files

If the logic of the program depends on raising the RECORD condition when a record of an incorrect length is found, you will have to write your own code to check for the record length and take the necessary action. This is because records of any length up to the maximum specified are allowed in VSAM data sets.

INDEXED files

You need to change programs using indexed (that is, ISAM) files only if you wish to change the way records are deleted. Because ISAM did not support deletion of records, records had to be 'logically' deleted, and programs had to check each record before processing it. The PL/I DELETE statement will physically remove a record from a VSAM KSDS, so you may want to change your programs to use this.

You should remove INDEXED and any other non-VSAM options from the file declaration, and replace them with ENV(VSAM). Specifically, the options related to ISAM files are:

```
ADDBUFF
HIGHINDEX
INDEXAREA
INDEXED
INDEXMULTIPLE
KEYLOC
NOWRITE
OFLTRACKS
```

REGIONAL(1) files

You can alter programs using REGIONAL(1) data sets to use VSAM relative-record data sets.

Remove REGIONAL(1) and any other non-VSAM ENVIRONMENT options from the file declaration and replace them with ENV(VSAM).

Also remove any checking for deleted records, because VSAM deleted records are not accessible to you.

Using several files in one VSAM data set

You can associate multiple files with one VSAM data set in the following ways:

- Use a common DLBL statement. You can use the TITLE option of the OPEN statement for this purpose, as described in "Associating data sets with files" on page 66.
- Use separate DLBL statements, and ensure that the DLBL statements reference the same data set name, or a path accessing the same underlying VSAM data set. If the DSN option of the ENVIRONMENT attribute is specified, VSAM will share control blocks based on a common data set name.

In both cases, PL/I creates one set of control blocks—an Access Method Control Block and a Request Parameter List (RPL)—for each file and does not provide for associating multiple RPLs with a single ACB. These control blocks are described in the VSAM publications, and normally need not concern you.

Multiple files can perform retrievals against a single data set with no difficulty. However, if one or more files perform updates, the following can occur:

- There is a risk that other files will retrieve down-level records. You can avoid this by having all files open with the UPDATE attribute.
- When more than one file is open with the UPDATE attribute, retrieval of any record in a control interval makes all other records in that control interval unavailable until the update is complete. This raises the ERROR condition with condition code 1027 if a second file tries to access one of the unavailable records. You could design your application to retry the retrieval after completion of the other file's data transmission, or you can avoid the error by not having two files associated with the same data set at one time.
- When one or more of the multiple files is an alternate index path, an update through an alternate index path might update the alternate index before the data record is written, resulting in a mismatch between index and data.

Using shared data sets

PL/I does not support cross-partition or cross-system sharing of data sets.

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 24.

Table 24 (Page 1 of 2). 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 OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEYTO(reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference);	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression)[3] KEYTO(reference)
	READ FILE(file-reference);[2]	EVENT(event-reference) and/or IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression)[3]
	READ FILE(file-reference)[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)[3]

Table 24 (Page 2 of 2). 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 UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression)[3] or KEYTO(reference)
	READ FILE(file-reference);[2]	EVENT(event-reference) and/or IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-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.

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 27 on page 155, 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 DLBL statement for the file contains the data set name ('file-id') 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.

Because the same program (in Figure 27) can be used for adding records to the data set, it is retained in a library by including the OPTION CATAL and PHASE statements. Figure 28 on page 155 shows the same program being reused to add more records to the data set.

```
// JOB    OPT9#7
// OPTION CATAL
 PHASE PGMA,*
// EXEC   IDCAMS,SIZE=128K
     DEFINE CLUSTER -
      (NAME(PLIVSAM.AJC1.BASE) -
      VOLUMES(VSE111) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2))
/*
// EXEC   IEL1AA,SIZE=128K
   CREATE: PROC OPTIONS(MAIN);

      DCL
        FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
        IN FILE RECORD INPUT ENV(F RECSIZE(80) MEDIUM(SYSIPT)),
        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;
/*
// EXEC   LNKEDT
// DLBL   FAMFILE,'PLIVSAM.AJC1.BASE',,VSAM
// EXEC   PGMA,SIZE=128K
FRED                    69          M
ANDY                    70          M
SUZAN                   72          F
/*
/&
```

Figure 27. Defining and loading an entry-sequenced data set (ESDS)

Updating an entry-sequenced data set

Figure 28 shows the addition of a new record on the end of an ESDS. This is done by executing again the program shown in Figure 27. The program uses a SEQUENTIAL OUTPUT file which is associated with the same data set ('PLIVSAM.AJC1.BASE') specified in the DEFINE command shown in Figure 27.

```
// JOB    OPT9#8
// DLBL   FAMFILE,'PLIVSAM.AJC1.BASE',,VSAM
// EXEC   PGMA,SIZE=128K
JANE                    75          F
/*
/&
```

Figure 28. Updating an ESDS

You can rewrite existing records in an ESDS, provided that the length of the record is not changed. You can use a SEQUENTIAL or KEYED SEQUENTIAL UPDATE file to do this. If you use keys, they can be the RBAs of the records or keys of an alternate index path.

Delete is not allowed for ESDS.

Key-sequenced and indexed entry-sequenced data sets

The statements and options allowed for indexed VSAM data sets are shown in Table 25. An indexed data set can be a KSDS with its prime index, or either a KSDS or an ESDS with an alternate index. Except where otherwise stated, the following description applies to all indexed VSAM data sets.

Table 25 (Page 1 of 3). 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[3]	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED[3]	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	EVENT(event-reference) and/or 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)[5]	KEY(expression)

Table 25 (Page 2 of 3). 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 UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	EVENT(event-reference) and/or IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEY(expression)
	DELETE FILE(file-reference);[5]	KEY(expression) and/or EVENT(event-reference)
DIRECT[4] INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT[4] INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
DIRECT[4] 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);[5]	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Table 25 (Page 3 of 3). 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[4] UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression);[5]	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the declaration.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);
3. Do not associate a SEQUENTIAL OUTPUT file with a data set accessed via an alternate index.
4. Do not associate a DIRECT file with a data set accessed via a nonunique alternate index.
5. DELETE statements are not allowed for a file associated with an ESDS accessed via an alternate index.

Loading a KSDS or indexed ESDS

When a KSDS is being loaded, you must open the associated file for KEYED SEQUENTIAL OUTPUT. You must present the records in ascending key order, and you must use the KEYFROM option. Note that you must use the prime index for loading the data set; you cannot load a VSAM data set via an alternate index.

If a KSDS already contains some records, and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can 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 29 on page 159 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 DLBL statement which uses the name given in the DEFINE command as the data set name.

```
// JOB    OPT9#12
// OPTION LINK
// EXEC   IDCAMS,SIZE=64K
   DEFINE CLUSTER -
     (NAME(PLIVSAM.AJC2.BASE) -
     VOLUMES(VSE111) -
     INDEXED -
     TRACKS(3 1) -
     KEYS(20 0) -
     RECORDSIZE(23 80))
/*
// EXEC   IEL1AA,SIZE=128K
 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;
/*
// EXEC   LNKEDT
// DLBL   DIREC,'PLIVSAM.AJC2.BASE',,VSAM
// EXEC   ,SIZE=128K
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
MANSON,K.J.         401
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 29. Defining and loading a key-sequenced data set (KSDS)

Using a SEQUENTIAL file to access a KSDS or indexed ESDS

You can open a SEQUENTIAL file that is used to access a KSDS with either the INPUT or the UPDATE attribute.

For READ statements without the KEY option, the records are recovered in ascending key order (or in descending key order if the BKWD option is used). You can obtain the key of a record recovered in this way by means of the KEYTO option.

If you use the KEY option, the record recovered by a READ statement is the one with the specified key. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, without respect to the position of any previous access. If you are accessing the data set via a unique index, the KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists on the data set. For a nonunique index, subsequent retrieval of records with the same key is in the order that they were added to the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the first record with the specified key; otherwise, it is the record that was accessed by the previous READ statement. When you rewrite a record using an alternate index, you can not change the primary key of the record.

Using a DIRECT file to access a KSDS or indexed ESDS

You can open a DIRECT file that is used to access an indexed VSAM data set with the INPUT, OUTPUT, or UPDATE attribute. Do not use a DIRECT file to access the data set via a nonunique index.

If you use a DIRECT OUTPUT file to add records to the data set, and if an attempt is made to insert a record with the same key as a record that already exists, the KEY condition is raised.

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

Figure 30 on page 161 shows one method by which a KSDS can be updated using the prime index.

```
// JOB    OPT9#13
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 DIRUPDT: PROC OPTIONS(MAIN);

       DCL DIREC FILE RECORD KEYED ENV(VSAM),
           ONCODE BUILTIN,
           OUTREC CHAR(23),
           NUMBER CHAR(3) DEF OUTREC POS(21),
           NAME CHAR(20) DEF OUTREC,
           CODE CHAR(1),
           EOF BIT(1) INIT('0'B);

       ON ENDFILE(SYSIN) EOF='1'B;

       ON KEY(DIREC) BEGIN;
          IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
                            ('NOT FOUND: ',NAME)(A(15),A);
          IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
                            ('DUPLICATE: ',NAME)(A(15),A);
       END;

       OPEN FILE(DIREC) DIRECT UPDATE;

       GET FILE(SYSIN) EDIT (NAME,  NUMBER, CODE)
                 (COLUMN(1), A(20), A(3),   A(1));
       DO WHILE (¬EOF);
          PUT FILE(SYSPRINT) SKIP EDIT
                  (' ',  NAME,  '#',  NUMBER, ' ',  CODE)
                  (A(1), A(20), A(1), A(3),   A(1), A(1));
          SELECT (CODE);
             WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
             WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
             WHEN('D') DELETE FILE(DIREC) KEY(NAME);
             OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
                ('INVALID CODE: ',NAME) (A(15),A);
          END;
          GET FILE(SYSIN) EDIT (NAME,  NUMBER, CODE)
                    (COLUMN(1), A(20), A(3),   A(1));
       END;

       CLOSE FILE(DIREC);
       PUT FILE(SYSPRINT) PAGE;
       OPEN FILE(DIREC) SEQUENTIAL INPUT;

       EOF='0'B;
       ON ENDFILE(DIREC) EOF='1'B;

       READ FILE(DIREC) INTO(OUTREC);
       DO WHILE(¬EOF);
       PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
       READ FILE(DIREC) INTO(OUTREC);
       END;
       CLOSE FILE(DIREC);
   END DIRUPDT;
/*
```

Figure 30 (Part 1 of 2). Updating a KSDS

```
// EXEC   LNKEDT
// DLBL   DIREC,'PLIVSAM.AJC2.BASE',,VSAM
// EXEC   ,SIZE=128K
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 30 (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

The program reads in the name, number, and code and takes action according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

The file is associated with the data set by a DLBL statement that uses the data set name PLIVSAM.AJC2.BASE defined in the Access Method Services DEFINE CLUSTER command in Figure 29 on page 159.

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. If the data had been presented in ascending key order (or even something approaching it), performance might have been improved by use of the SKIP ENVIRONMENT option. 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. VSAM provides three methods of insertion as shown in Table 26 on page 163.

Table 26. VSAM methods of insertion into a KSDS

Method	Requirements	Freespace	When written onto data set	PL/I attributes required
Sequential	Keys in sequence	Kept	Only when necessary	KEYED SEQUENTIAL UPDATE
Skip-sequential	Keys in sequence	Used	Only when necessary	KEYED SEQUENTIAL UPDATE ENV(VSAM SKIP)
Direct	Keys in any order	Used	After every statement	DIRECT

SKIP means that you must follow the sequence but that you can omit records. You do not need to maintain absolute sequence or order if sequential or skip is used. The PL/I routines determine which type of request to make to VSAM for each statement, first checking the keys to determine which would be appropriate. The retention of free space ensures that the structure of the data set at the point of mass sequential insertion is not destroyed, enabling you to make further normal alterations in that area without loss of performance.

Alternate indexes for KSDSs or indexed ESDSs

Alternate indexes allow you to access KSDSs or indexed ESDSs in various ways, using either unique or nonunique keys.

Unique key alternate index path

Figure 31 on page 164 shows the creation of a unique key alternate index path for the ESDS defined and loaded in Figure 27 on page 155. Using this path, the data set is indexed by the name of the child in the first 15 bytes of the record.

Three Access Method Services commands are used. These are:

DEFINE ALTERNATEINDEX
defines the alternate index as a data set to VSAM.

BLDINDEX
places the pointers to the relevant records in the alternate index.

DEFINE PATH
defines an entity that can be associated with a PL/I file in a DLBL statement.

DLBL statements are required for the INFILE and OUTFILE operands of BLDINDEX. Make sure that you specify the correct names at the various points.

```
// JOB    OPT9#9
// EXEC   IDCAMS,SIZE=128K
         DEFINE ALTERNATEINDEX -
           (NAME(PLIVSAM.AJC1.ALPHIND) -
            VOLUMES(VSE111) -
            TRACKS(4 1) -
            KEYS(15 0) -
            RECORDSIZE(20 40) -
            UNIQUEKEY -
           RELATE(PLIVSAM.AJC1.BASE))
/*
// DLBL   DD1,'PLIVSAM.AJC1.BASE',,VSAM
// DLBL   DD2,'PLIVSAM.AJC1.ALPHIND',,VSAM
// EXEC   IDCAMS,SIZE=128K
         BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
         DEFINE PATH -
           (NAME(PLIVSAM.AJC1.ALPHPATH) -
           PATHENTRY(PLIVSAM.AJC1.ALPHIND))
/*
/&
```

Figure 31. Creating a unique key alternate index path for an ESDS

Nonunique key alternate index path

Figure 32 shows the creation of a nonunique key alternate index path for an ESDS. The alternate index enables the data to be selected by the sex of the children. This enables the girls or the boys to be accessed separately and every member of each group to be accessed by use of the key.

The three Access Method Services commands and the DLBL statement are as described in "Unique key alternate index path" on page 163. The fact that the index has nonunique keys is specified by the use of the NONUNIQUEKEY operand. When creating an index with nonunique keys, be careful to ensure that the RECORDSIZE you specify is large enough. In a nonunique alternate index, each alternate index record contains pointers to all the records that have the associated alternate index key. The pointer takes the form of an RBA for an ESDS and the prime key for a KSDS. When a large number of records might have the same key, a large record is required.

```
// JOB    OPT9#10
// EXEC   IDCAMS,SIZE=128K
     /*care must be taken with record size */
     DEFINE ALTERNATEINDEX -
       (NAME(PLIVSAM.AJC1.SEXIND) -
        VOLUMES(VSE111) -
        TRACKS(4 1) -
        KEYS(1 37) -
        NONUNIQUEKEY -
        RELATE(PLIVSAM.AJC1.BASE) -
        RECORDSIZE(20 400))
/*
// DLBL   DD1,'PLIVSAM.AJC1.BASE',,VSAM
// DLBL   DD2,'PLIVSAM.AJC1.SEXIND',,VSAM
// EXEC   IDCAMS,SIZE=128K
    BLDINDEX  INFILE(DD1)  OUTFILE(DD2)
    DEFINE PATH -
      (NAME(PLIVSAM.AJC1.SEXPATH) -
       PATHENTRY(PLIVSAM.AJC1.SEXIND))
/*
/&
```

Figure 32. Creating a nonunique key alternate index path for an ESDS

Figure 33 on page 165 shows the creation of a unique key alternate index path for a KSDS. The data set is indexed by the telephone number, enabling the number to be used as a key to discover the name of the person on that extension. The fact that keys are to be unique is specified by UNIQUEKEY. Also, the data set will be able to be listed in numerical order to show which numbers are not used. Three Access Method Services commands are used:

DEFINE ALTERNATEINDEX
defines the data set that will hold the alternate index data.

BLDINDEX
places the pointers to the relevant records in the alternate index.

DEFINE PATH
defines the entity that can be associated with a PL/I file in a DLBL statement.

DLBL statements are required for the INFILE and OUTFILE of BLDINDEX. Be careful not to confuse the names involved.

```
// JOB    OPT9#14
// EXEC   IDCAMS,SIZE=128K
        DEFINE ALTERNATEINDEX -
          (NAME(PLIVSAM.AJC2.NUMIND) -
           VOLUMES(VSE111) -
           TRACKS(4 4) -
           KEYS(3 20) -
           RELATE(PLIVSAM.AJC2.BASE) -
           UNIQUEKEY -
           RECORDSIZE(24 48))
/*
// DLBL   DD1,'PLIVSAM.AJC2.BASE',,VSAM
// DLBL   DD2,'PLIVSAM.AJC2.NUMIND',,VSAM
// EXEC   IDCAMS,SIZE=128K
        BLDINDEX  INFILE(DD1)  OUTFILE(DD2)

        DEFINE PATH -
          (NAME(PLIVSAM.AJC2.NUMPATH) -
           PATHENTRY(PLIVSAM.AJC2.NUMIND))
/*
/&
```

Figure 33. Creating an alternate index path for a KSDS

When creating an alternate index with a unique key, you should ensure that no further records could be included with the same alternative key. In practice, a unique key alternate index would not be entirely satisfactory for a telephone directory as it would not allow two people to have the same number. Similarly, the prime key would prevent one person having two numbers. A solution would be to have an ESDS with two nonunique key alternate indexes, or to restructure the data format to allow more than one number per person and to have a nonunique key alternate index for the numbers. See Figure 31 on page 164 for an example of creation of an alternate index with nonunique keys.

Detecting nonunique alternate index keys

If you are accessing a VSAM data set by means of an alternate index path, the presence of nonunique keys can be detected by means of the SAMEKEY built-in function. After each retrieval, SAMEKEY indicates whether any further records exist with the same alternate index key as the record just retrieved. Hence, it is possible to stop at the last of a series of records with nonunique keys without having to read beyond the last record. SAMEKEY (file-reference) returns '1'B if

the input/output statement has completed successfully and the accessed record is followed by another with the same key; otherwise, it returns '0'B.

Using alternate indexes with ESDSs

Figure 34 on page 167 shows the use of alternate indexes and backward reading on an ESDS. The program has four files:

BASEFLE reads the base data set forward.
BACKFLE reads the base data set backward.
ALPHFLE is the alphabetic alternate index path indexing the children by name.
SEXFILE is the alternate index path that corresponds to the sex of the children.

The JCL contains DLBL statements for all the files. They connect BASEFLE and BACKFLE to the base data set by specifying its data set name, and connect ALPHFLE and SEXFLE by specifying the names of the paths given in Figure 31 on page 164 and Figure 32 on page 164.

The program uses SEQUENTIAL files to access the data and print it first in the normal order, then in the reverse order. Then a DIRECT file is used to read the data associated with an alternate index key in the unique alternate index.

Finally, at the label SPRINT, a KEYED SEQUENTIAL file is used to print a list of the females in the family, using the nonunique key alternate index path. The SAMEKEY built-in function is used to read all the records with the same key. The names of the females will be accessed in the order in which they were originally entered. This will happen whether the file is read forward or backward. For a nonunique key path, the BKWD option only affects the order in which the keys are read; the order of items with the same key remains the same as it is when the file is read forward.

Deletion: At the end of the example, the Access Method Services DELETE command is used to delete the base data set. When this is done, the associated alternate indexes and paths will also be deleted.

Using alternate indexes with KSDSs

Figure 35 on page 169 shows the use of a path with a unique alternate index key to update a KSDS and then to access and print it in the order of the alternate index.

The alternate index path is associated with the PL/I file by a DLBL statement that specifies the name of the path (PLIVSAM.AJC2.NUMPATH, given in the DEFINE PATH command in Figure 33 on page 165) as its data set name.

In the first section of the program, a DIRECT OUTPUT file is used to insert a new record using the alternate index key. Note that any alteration made with an alternate index must not alter the prime key or the alternate index key of access of an existing record. Also, the alteration must not add a duplicate key in the prime index or any unique key alternate index.

In the second section of the program the data set is read in the order of the alternate index keys using a SEQUENTIAL INPUT file. It is then printed onto SYSPRINT.

```
// JOB    OPT9#15
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
  READIT: PROC OPTIONS(MAIN);
    DCL   BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
                 /*File to read base data set forward */
          BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
                 /*File to read base data set backward */
          ALPHFLE FILE DIRECT INPUT ENV(VSAM),
          /*File to access via unique alternate index path */
          SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
           /*File to access via nonunique alternate index path */
          STRING CHAR(80),  /*String to be read into */
          1 STRUC DEF (STRING),
            2 NAME CHAR(25),
            2 DATE_OF_BIRTH CHAR(2),
            2 FILL CHAR(10),
            2 SEX CHAR(1);
            DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;
            DCL EOF BIT(1) INIT('0'B);

    /*Print out the family eldest first*/

    ON ENDFILE(BASEFLE) EOF='1'B;
    PUT EDIT('FAMILY ELDEST FIRST')(A);
    READ FILE(BASEFLE) INTO (STRING);
    DO WHILE(¬EOF);
       PUT SKIP EDIT(STRING)(A);
       READ FILE(BASEFLE) INTO (STRING);
    END;
    CLOSE FILE(BASEFLE);
    PUT SKIP(2);
     /*Close before using data set from other file not
       necessary but good practice to prevent potential
       problems*/

    EOF='0'B;
    ON ENDFILE(BACKFLE) EOF='1'B;
    PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);
    READ FILE(BACKFLE) INTO(STRING);
    DO WHILE(¬EOF);
      PUT SKIP EDIT(STRING)(A);
      READ FILE(BACKFLE) INTO (STRING);
    END;
    CLOSE FILE(BACKFLE);
    PUT SKIP(2);

    /*Print date of birth of child specified in the file
      SYSIN*/
    ON KEY(ALPHFLE) BEGIN;
       PUT SKIP EDIT
          (NAMEHOLD,' NOT A MEMBER OF THE SMITH FAMILY') (A);
       GO TO SPRINT;
    END;
```

Figure 34 (Part 1 of 2). Alternate index paths and backward reading with an ESDS

```
    EOF='0'B;
    ON ENDFILE(SYSIN) EOF='1'B;
    GET SKIP EDIT(NAMEHOLD)(A(25));
    DO WHILE(¬EOF);
          READ FILE(ALPHFLE) INTO (STRING) KEY(NAMEHOLD);
          PUT SKIP (2) EDIT(NAMEHOLD,' WAS BORN IN ',
              DATE_OF_BIRTH)(A,X(1),A,X(1),A);
          GET SKIP EDIT(NAMEHOLD)(A(25));
    END;
 SPRINT:
    CLOSE FILE(ALPHFLE);
    PUT SKIP(1);

   /*Use the alternate index to print out all the females in the
     family*/
       ON ENDFILE(SEXFILE) GOTO FINITO;
       PUT SKIP(2) EDIT('ALL THE FEMALES')(A);
       READ FILE(SEXFILE) INTO (STRING) KEY('F');
       PUT SKIP EDIT(STRING)(A);
       DO WHILE(SAMEKEY(SEXFILE));
          READ FILE(SEXFILE) INTO (STRING);
          PUT SKIP EDIT(STRING)(A);
       END;

 FINITO:
     END;
/*
// EXEC   LNKEDT
// DLBL   BASEFLE,'PLIVSAM.AJC1.BASE',,VSAM
// DLBL   BACKFLE,'PLIVSAM.AJC1.BASE',,VSAM
// DLBL   ALPHFLE,'PLIVSAM.AJC1.ALPHPATH',,VSAM
// DLBL   SEXFILE,'PLIVSAM.AJC1.SEXPATH',,VSAM
// EXEC   ,SIZE=128K
ANDY
/*
// EXEC   IDCAMS,SIZE=64K
     DELETE -
          PLIVSAM.AJC1.BASE
/*
/&
```

Figure 34 (Part 2 of 2). Alternate index paths and backward reading with an ESDS

```
// JOB    OPT9#16
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
    ALTER: PROC OPTIONS(MAIN);
       DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
           NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
           IN FILE RECORD INPUT ENV(F RECSIZE(80) MEDIUM(SYSIPT)),
           STRING CHAR(80),
           NAME CHAR(20) DEF STRING,
           NUMBER CHAR(3) DEF STRING POS(21),
           DATA CHAR(23) DEF STRING,
           EOF BIT(1) INIT('0'B);

      ON KEY (NUMFLE1) BEGIN;
         PUT SKIP EDIT('DUPLICATE NUMBER')(A);
      END;

      ON ENDFILE(IN) EOF='1'B;

      READ FILE(IN) INTO (STRING);
      DO WHILE(¬EOF);
         PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
         WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
         READ FILE(IN) INTO (STRING);
      END;

      CLOSE FILE(NUMFLE1);

      EOF='0'B;
      ON ENDFILE(NUMFLE2) EOF='1'B;

      PUT SKIP(3) EDIT('****PHONE DIRECTORY (NUMERICAL)****')(A);

      READ FILE(NUMFLE2) INTO (STRING);
      DO WHILE(¬EOF);
        PUT SKIP EDIT(DATA)(A);
        READ FILE(NUMFLE2) INTO (STRING);
      END;

      PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
    END;
/*
// EXEC   LNKEDT
// DLBL   NUMFLE1,'PLIVSAM.AJC2.NUMPATH',,VSAM
// DLBL   NUMFLE2,'PLIVSAM.AJC2.NUMPATH',,VSAM
// EXEC   ,SIZE=128K
RIERA,L             123
/*
// EXEC   IDCAMS,SIZE=64K
    DELETE -
         PLIVSAM.AJC2.BASE
/*
/&
```

Figure 35. Using a unique alternate index path to access a KSDS

Relative-record and variable-length relative-record data sets

The statements and options allowed for VSAM relative-record data sets (RRDS) and variable-length relative-record data sets (VRDS) are shown in Table 27.

Table 27 (Page 1 of 3). Statements and options allowed for loading and accessing RRDS and VRDS

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

Table 27 (Page 2 of 3). Statements and options allowed for loading and accessing RRDS and VRDS

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

Table 27 (Page 3 of 3). Statements and options allowed for loading and accessing RRDS and VRDS

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

Notes:

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, your declaration must also include the attribute KEYED.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);

Loading an RRDS or VRDS

When an RRDS or VRDS is being loaded, you must open the associated file for OUTPUT. You can 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 140).

For a SEQUENTIAL OUTPUT file, use WRITE statements with or without the KEYFROM option. If you specify the KEYFROM option:

- For an RRDS, the record is placed in the specified slot; if you omit it, the record is placed in the slot following the current position.
- For a VRDS, the record is written with the specified record number as the key; if you omit it, the record is written with a key equal to the current record number plus one.

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 or VRDS 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 36 on page 174, the data set is defined with a DEFINE CLUSTER command and given the name PLIVSAM.AJC3.BASE.

The NUMBERED keyword indicates that it is a relative-record data set. The RECORDSIZE keyword distinguishes between an RRDS and a VRDS by the specification of the average and maximum record lengths. If these are the same (as in this example), the data set is an RRDS. If they are different, the data set is a VRDS. In the PL/I program, it is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

The PL/I file is associated with the data set by the DLBL statement, which uses as its data set name the name given in the DEFINE CLUSTER command.

```
// JOB    OPT9#17
// OPTION LINK
// EXEC   IDCAMS,SIZE=128K
         DEFINE CLUSTER -
             (NAME(PLIVSAM.AJC3.BASE) -
             VOLUMES(VSE111) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20))
/*
// EXEC   IEL1AA,SIZE=128K
 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;
/*
// EXEC   LNKEDT
// DLBL   NOS,'PLIVSAM.AJC3.BASE',,VSAM
// EXEC   ,SIZE=128K
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
MANSON,K.J.         45
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 36. Defining and loading a relative-record data set

Using a SEQUENTIAL file to access a relative-record data set

You can open a SEQUENTIAL file that is used to access an RRDS or VRDS 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. For RRDS, any empty slots in the data set are skipped (there are no empty slots for VRDS).

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:

- For RRDS, 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.
- For VRDS, an attempt is made to write the record with a key equal to the current record number plus one. The ERROR condition is raised if this record number already exists.

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 or VRDS

A DIRECT file used to access an RRDS or VRDS 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 37 on page 176 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. When reading, there is no need to check whether records are empty, because the empty records are not available under VSAM. (For VRDS, records either exist or do not exist; there are no “empty” VRDS records.)

In the second half of the program, 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 DLBL statement that specifies PLIVSAM.AJC3.BASE as its data set name, the name given in the DEFINE CLUSTER command in Figure 36 on page 174.

At the end of the example, the DELETE command is used to delete the data set.

```
// JOB    OPT9#18
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 /* 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.                 */
 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;
       GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
          (COLUMN(1),A(20),A(2),A(2),A(1));
       END;
       CLOSE FILE(NOS);
       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;
/*
```

Figure 37 (Part 1 of 2). Updating a relative-record data set

```
// EXEC   LNKEDT
// DLBL   NOS,'PLIVSAM.AJC3.BASE',,VSAM
// EXEC   ,SIZE=128K
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
/*
// EXEC   IDCAMS,SIZE=128K
      DELETE -
          PLIVSAM.AJC3.BASE
/*
/&
```

Figure 37 (Part 2 of 2). Updating a relative-record data set

Part 4. Improving your program

Chapter 9. Examining and tuning compiled modules

This chapter discusses how to obtain static information about your compiled program or other object modules of interest either during execution of your program or at any time. Specifically, it discusses:

- How to turn hooks on prior to execution by calling IBMBHKS (see "Activating hooks in your compiled program using IBMBHKS")
- How to call the Static Information Retrieval service IBMBSIR to retrieve static information about compiled modules (see "Obtaining static information about compiled modules using IBMBSIR" on page 182)
- How to call the Hook Information Retrieval service IBMBHIR to obtain static information relative to hooks that are executed during your program's run (see "Obtaining static information as hooks are executed using IBMBHIR" on page 186)
- How to use IBMBHKS, IBMBSIR, and IBMBHIR via the hook exit in CEEBINT to examine your program's run-time behavior (see "Examining your program's run-time behavior" on page 187).

These services are useful if you want to do any of the following:

- Examine and fine tune your program's run-time behavior by, for example, checking which statements, blocks, paths, labels, or calls are visited most
- Perform function tracing during your program's execution
- Examine CPU timing characteristics of your program's execution
- Find out information about any object module such as:
  - What options it was compiled with
  - Its size
  - The number and location of blocks in the module
  - The number and location of hooks in the module
  - The number and addresses of external entries in the module

These services are available in batch and CICS environments.

For information on how to establish the hook exit in CEEBINT, see the *LE/VSE Programming Guide.*

Activating hooks in your compiled program using IBMBHKS

The callable service IBMBHKS is provided to turn hooks on and off without the use of a debugging tool. It is available in batch and CICS environments.

The IBMBHKS programming interface

You can declare IBMBHKS in a PL/I program as follows:

```
DECLARE IBMBHKS EXTERNAL ENTRY( FIXED BIN(31,0), FIXED BIN(31,0) );
```

and invoke it with the following PL/I CALL statement:

```
CALL IBMBHKS( Function_code, Return_code );
```

The possible function codes are:

1 Turn on statement hooks
-1 Turn off statement hooks
2 Turn on block entry hooks
-2 Turn off block entry hooks
3 Turn on block exit hooks
-3 Turn off block exit hooks
4 Turn on path hooks
-4 Turn off path hooks
5 Turn on label hooks
-5 Turn off label hooks
6 Turn on before-call hooks
-6 Turn off before-call hooks
7 Turn on after-call hooks
-7 Turn off after-call hooks.

The possible return codes are:

0 Successful
12 The debugging tool is active
16 Invalid function code passed.

Note: Turning on or off statement hooks or path hooks also turns on or off respectively block entry and block exit hooks. The reverse, however, is not true.

Warning

This service is meant to be used with the hook exit. It is an error to use IBMBHKS to turn on hooks when a hook exit has not been established in CEEBINT, and unpredictable results will occur in this case.

For examples of possible uses of IBMBHKS see "Examining your program's run-time behavior" on page 187.

Obtaining static information about compiled modules using IBMBSIR

IBMBSIR is a Static Information Retrieval module that lets you determine static information about PL/I modules compiled with the TEST option. You can use it to interrogate static information about a compiled module (to find out, for example, what options it was compiled with), or you can use it recursively as part of a run-time monitoring process.

It is available in batch and CICS environments.

The IBMBSIR programming interface

You invoke IBMBSIR with a PL/I CALL passing the address of a control block containing the following elements:

```
DECLARE
  1 SIR_DATA        BASED,
    2 SIR_FNCCODE   FIXED BIN(31), /* Function code             */
    2 SIR_RETCODE   FIXED BIN(31), /* Return code               */
    2 SIR_ENTRY     ENTRY,         /* Entry variable for module */
    2 SIR_MOD_DATA  POINTER,       /* Addr of module_data       */
    2 SIR_A_DATA    POINTER,       /* Addr of data for fnc code */
    2 SIR_END       CHAR(0);
```

The items in the control block have the following meanings:

SIR_FNCCODE
Function code specifying the type of information that is desired, according to the following definitions:

1 Fill in the compile-time options information and the count of blocks in the module information control block (see the MODULE_OPTIONS array and MODULE_BLOCKS in Figure 38 on page 185).

2 Same function as **1** but also fill in the module's size, MODULE_SIZE, in the module information control block.

3 Fill in all information specified in the module information control block; namely, compile-time options, count of blocks, module size, and counts of statements, paths, and external entries declared explicitly or implicitly.

Before invoking IBMBSIR with a function code of 4, you must have already invoked it with one of the first three function codes:

4 Fill in the information specified in the block information control block. The layout of this control block is given in Figure 39 on page 185.

BLOCK_NAME_LEN can be zero for unlabeled BEGIN blocks.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields BLOCK_DATA and BLOCK_COUNT.

Before invoking IBMBSIR with any of the following function codes, you must have already invoked it with a function code of 3:

5 Fill in the hook information block for all statement hooks. The layout of this control block is given in Figure 40 on page 186.

Note that statement hooks include block entry and exit hooks.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields HOOK_DATA and HOOK_COUNT.

HOOK_IN_LINE will be zero for all programs compiled with the STMT compile-time option, and for programs compiled with the NUMBER compile-time option, it will be nonzero only when a statement is one of a multiple in a source line.

HOOK_OFFSET is always the offset of the hook from the primary entry point for the module, not the offset of the hook from the primary entry point for the block in which the hook occurs.

6 Fill in the hook information control block (see Figure 40 on page 186) for all path hooks.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields HOOK_DATA and HOOK_COUNT.

7 Fill in the external entry information control block. The layout of this control block is given in Figure 41 on page 186.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields EXTS_DATA and EXTS_COUNT.

EXTS_EPA will give the entry point address for a module declared explicitly or implicitly in the program. It will be zero if the module has not been resolved at link-edit time.

Note: For all of the function codes you must also supply the SIR_ENTRY and SIR_MOD_DATA parameters described below.

For function codes 4, 5, 6, and 7 you must supply the SIR_A_DATA parameter described below.

SIR_RETCODE
The return code. This will have the following values:

0 Successful.
4 Module not compiled with appropriate TEST option.
8 Module not PL/I or not compiled with TEST option.
12 Invalid parameters passed.
16 Unknown function code.

SIR_ENTRY
The main entry point to your module.

SIR_MOD_DATA
A pointer to the module information control block, shown in Figure 38 on page 185.

SIR_A_DATA
is a pointer to the block information control block, the hook information control block, or the external entries information control block, depending on which function code you are using.

The following figures show the layout of the control blocks:

Figure 39 on page 185 shows the block information control block.
Figure 40 on page 186 shows the hook information control block.
Figure 41 on page 186 shows the external entries information control block.

This field must be zero if you specify a function code of 1, 2, or 3. If you specify function codes 4, 5, 6, or 7, this field must point to the applicable control block:

Function Code	*Set Pointer to*
4	Block information control block
5 or 6	Hook information control block
7	External entries information control block.

```
DECLARE
  1 MODULE_DATA     BASED,
                                    /*                              */
    2 MODULE_LEN    FIXED BIN(31),  /* = Stg(Module_data)           */
                                    /*                              */
    2 MODULE_OPTIONS,               /* Compile time options         */
                                    /*                              */
      3 MODULE_GENERAL_OPTIONS,     /*                              */
        4 MODULE_STMTNO  BIT(01),   /*   Stmt number table does     */
                                    /*   not exist                  */
        4 MODULE_GONUM   BIT(01),   /*   Table has GONUMBER form    */
        4 MODULE_CMPATV1 BIT(01),   /*   compiled with CMPAT(V1)    */
        4 MODULE_GRAPHIC BIT(01),   /*   compiled with GRAPHIC      */
        4 MODULE_OPT     BIT(01),   /*   compiled with OPTIMIZE     */
        4 MODULE_INTER   BIT(01),   /*   compiled with INTERRUPT    */
        4 MODULE_GEN1X   BIT(02),   /*   Reserved                   */
                                    /*                              */
      3 MODULE_GENERAL_OPTIONS2,    /*                              */
        4 MODULE_GEN2X   BIT(08),   /*   Reserved                   */
                                    /*                              */
      3 MODULE_TEST_OPTIONS,        /*                              */
        4 MODULE_TEST    BIT(01),   /*   compiled with TEST         */
        4 MODULE_STMT    BIT(01),   /*      with STMT  suboption    */
        4 MODULE_PATH    BIT(01),   /*      with PATH  suboption    */
        4 MODULE_BLOCK   BIT(01),   /*      with BLOCK suboption    */
        4 MODULE_TESTX   BIT(03),   /*   Reserved                   */
        4 MODULE_SYM     BIT(01),   /*      with SYM   suboption    */
                                    /*                              */
      3 MODULE_SYS_OPTIONS,         /*                              */
        4 MODULE_CMS     BIT(01),   /*   SYSTEM(CMS)                */
        4 MODULE_CMSTP   BIT(01),   /*   SYSTEM(CMSTPL)             */
        4 MODULE_MVS     BIT(01),   /*   SYSTEM(MVS)                */
        4 MODULE_TSO     BIT(01),   /*   SYSTEM(TSO)                */
        4 MODULE_CICS    BIT(01),   /*   SYSTEM(CICS)               */
        4 MODULE_IMS     BIT(01),   /*   SYSTEM(IMS)                */
        4 MODULE_VSE     BIT(01),   /*   SYSTEM(VSE)                */
        4 MODULE_SYSX    BIT(01),   /*   Reserved                   */
                                    /*                              */
    2 MODULE_BLOCKS FIXED BIN(31),  /* Count of blocks              */
    2 MODULE_SIZE   FIXED BIN(31),  /* Size of module               */
    2 MODULE_SHOOKS FIXED BIN(31),  /* Count of stmt hooks          */
    2 MODULE_PHOOKS FIXED BIN(31),  /* Count of path hooks          */
    2 MODULE_EXTS   FIXED BIN(31),  /* Count of external entries    */

    2 MODULE_DATA_END    CHAR(0);
```

Figure 38. Module information control block

```
DECLARE

  1 BLOCK_TABLE    BASED,

    2 BLOCK_DATA       POINTER,         /*  Addr of BLOCK_INFO      */
    2 BLOCK_COUNT      FIXED BIN(31),   /*  Count of blocks         */

    2 BLOCK_INFO( BLOCKS REFER(BLOCK_COUNT) ),

      3 BLOCK_OFFSET   FIXED BIN(31),   /* Offset of block entry    */
      3 BLOCK_SIZE     FIXED BIN(31),   /* Size of block            */
      3 BLOCK_LEVEL    FIXED BIN(15),   /* Block nesting level      */
      3 BLOCK_PARENT   FIXED BIN(15),   /* Index for parent block   */
      3 BLOCK_CHILD    FIXED BIN(15),   /* Index for first child    */
      3 BLOCK_SIBLING  FIXED BIN(15),   /* Index for next sibling   */
      3 BLOCK_NAME_LEN FIXED BIN(15),   /* Length of block name     */
      3 BLOCK_NAME_STR CHAR(34),        /* Block name               */

    2 BLOCK_TABLE_END    CHAR(0);
```

Figure 39. Block information control block

```
DECLARE

  1 HOOK_TABLE     BASED,

    2 HOOK_DATA        POINTER,         /*  Addr of HOOK_INFO       */
    2 HOOK_COUNT       FIXED BIN(31),   /*  Count of hooks          */

    2 HOOK_INFO( HOOKS REFER(HOOK_COUNT) ),

      3 HOOK_OFFSET    FIXED BIN(31),    /* Offset of hook          */
      3 HOOK_NO        FIXED BIN(31),    /* Stmt number for hook    */
      3 HOOK_IN_LINE   FIXED BIN(15),    /* Stmt number within line */
      3 HOOK_RESERVED  FIXED BIN(15),    /* Reserved                */
      3 HOOK_TYPE      FIXED BIN(15),    /* Hook type (=%PATHCODE)  */
      3 HOOK_BLOCK     FIXED BIN(15),    /* Block number for hook   */

    2 HOOK_TABLE_END   CHAR(0);
```

Figure 40. Hook information control block

```
DECLARE

  1 EXTS_TABLE     BASED,

    2 EXTS_DATA        POINTER,          /*  Addr of EXTS_INFO      */
    2 EXTS_COUNT       FIXED BIN(31),    /*  Count of entries       */

    2 EXTS_INFO( EXTS REFER(EXTS_COUNT) ),

      3 EXTS_EPA       POINTER,          /*  EPA for entry          */

    2 EXTS_TABLE_END   CHAR(0);
```

Figure 41. External entries information control block

For examples of possible uses of IBMBSIR see "Examining your program's run-time behavior" on page 187.

Obtaining static information as hooks are executed using IBMBHIR

IBMBHIR is a Hook Information Retrieval module that lets you determine static information about hooks executed in modules compiled with the TEST option. It is available in batch and CICS environments.

The IBMBHIR programming interface

You invoke IBMBHIR with a PL/I CALL passing, in order, the address of the control block shown below, the value of register 13 when the hook was executed, and the address of the hook that was executed. (These last two items are also the last two items passed to the hook exit.)

```
DECLARE
  1 HIR_DATA         BASED(HIR_PARMS),
    2 HIR_STG        FIXED BIN(31),    /* Size of this control block */
    2 HIR_EPA        POINTER,          /* Addr of module entry point */
    2 HIR_LANG_CODE  BIT(8),           /* Language code              */
    2 HIR_PATH_CODE  BIT(8),           /* Path code for hook         */
    2 HIR_NAME_LEN   FIXED BIN(15),    /* Length of module name      */
    2 HIR_NAME_ADDR  POINTER,          /* Addr of module name        */
    2 HIR_BLOCK      FIXED BIN(31),    /* Block count                */
    2 HIR_END        CHAR(0);
```

These parameters, upon return from IBMBHIR, supply:

- Information about the module in which the hook was executed:

  HIR_EPA
  The primary entry point address of the module (you could use this value as a parameter to IBMBSIR to obtain more data)

  HIR_LANG_CODE
  The programming language used to compile the module ('0A'X for PL/I)

  HIR_NAME_LEN
  The length of the name of the module

  HIR_NAME_ADDR
  The address of a nonvarying string containing the module name

- Information about the block in which the hook was executed:

  HIR_BLOCK
  the block number for that block

- Information about the hook itself:

  HIR_PATH_CODE
  the %PATHCODE value associated with the hook.

The next section contains an example of one of the possible uses of IBMBHIR.

Examining your program's run-time behavior

This section shows some practical ways of using the services discussed in the first part of this chapter (IBMBHKS, IBMBSIR, and IBMBHIR) via the hook exit in CEEBINT to monitor your program's run-time behavior. In particular, three sample facilities are presented, which demonstrate, respectively, how you can:

- Examine code coverage (see "Sample facility 1: Examining code coverage" on page 187)
- Perform function tracing (see "Sample facility 2: Performing function tracing" on page 200)
- Analyze CPU-time usage (see page "Sample facility 3: Analyzing CPU-time usage" on page 204).

For each facility, the overall setup is outlined briefly, the output is given, and the source code for the facility is shown.

Sample facility 1: Examining code coverage

The following example programs show how to establish a rather simple hook exit to report on code coverage.

Overall setup

This facility consists of two programs: CEEBINT and HOOKUP. The CEEBINT module is coded so that:

- A hook exit to the HOOKUP program is established
- Calls to IBMBHKS are made to set hooks prior to execution.

At run time, whenever HOOKUP gains control (via the established hook exit), it calls IBMBSIR to obtain code coverage information on the MAIN procedure and those linked with it.

Note: The CEEBINT routine uses a recursive routine `Add_Module_to_List` to locate and save information on the MAIN module and all the modules linked with it. Before this routine is recursively invoked, a check should be made to see if the module to be added has already been added. If such a check is not made, the subroutine could call itself endlessly.

The SPROG suboption of the LANGLVL compile-time option is specified in order to enable the adding to a pointer that takes place in CEEBINT and the comparing of two pointers that takes place in HOOKUP.

Output generated

The output given in Figure 42 is generated during execution of a PL/I program called KNIGHT. The KNIGHT program's function is to determine the moves a knight must make to land on each square of a chess board only once.

The output is created by the HOOKUP program as employed in this facility.

```
Post processing


Data for block KNIGHT

     Statement  Type                      Visits    Percent

             1  block entry                    1     0.0264
            14  before call                    1     0.0264
            14  after call                     1     0.0264
            21  start of do loop              63     1.6662
            23  start of do loop             504    13.3298
            25  if-true                      229     6.0565
            25  if-true                       91     2.4067
            29  if-false                     138     3.6498
            30  if-false                     275     7.2732
            32  if-true                       63     1.6662
            33  start of do loop             504    13.3298
            34  if-true                      224     5.9243
            35  if-false                     280     7.4054
            42  if-false                       0     0.0000
            44  start of do loop               8     0.2115
            65  block exit                     1     0.0264

Data for block INITIALIZE_RANKINGS

     Statement  Type                      Visits    Percent

            48  block entry                    1     0.0264
            50  start of do loop              12     0.3173
            51  start of do loop             144     3.8085
            52  if-true                       80     2.1158
            53  if-false                      64     1.6926
            56  start of do loop               8     0.2115
            57  start of do loop              64     1.6926
            58  start of do loop             512    13.5413
            59  if-true                      176     4.6548
            60  if-false                     336     8.8865
            64  block exit                     1     0.0264
```

Figure 42. Code coverage produced by sample facility 1

Source code

The source code for Sample Facility 1 is distributed with the PL/I VSE product, and is shown here for easy reference. The sample routines are:

Routine name	Distributed as	Shown in
CEEBINT	IELS1INT.P	Figure 43 on page 190
HOOKUP	IELS1HKP.P	Figure 44 on page 196

```
*PROCESS FLAG(I) GOSTMT STMT SOURCE;
*PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
 CEEBINT: Proc( Number, RetCode, RsnCode, FncCode, A_Main,
                UserWd, A_Exits )
                options(reentrant) reorder;

   Dcl Number     fixed bin(31);      /* Number of args = 7        */
   Dcl RetCode    fixed bin(31);      /* Return Code = 0           */
   Dcl RsnCode    fixed bin(31);      /* Reason Code = 0           */
   Dcl FncCode    fixed bin(31);      /* Function Code = 1         */
   Dcl A_Main     pointer;            /* Address of Main Routine   */
   Dcl UserWd     fixed bin(31);      /* User Word                 */
   Dcl A_Exits    pointer;            /* A(Exits list)             */

   Declare A_exit_list       pointer;

   Declare
     1 Exit_list         based(A_exit_list),
       2 Exit_list_count    fixed bin(31),
       2 Exit_list_slots,
         3 Exit_for_hooks   pointer,
       2 Exit_list_end      char(0);

   Declare
     1 Hook_exit_block   based(Exit_for_hooks),
       2 Hook_exit_len      fixed bin(31),
       2 Hook_exit_rtn      pointer,
       2 Hook_exit_fnccode  fixed bin(31),
       2 Hook_exit_retcode  fixed bin(31),
       2 Hook_exit_rsncode  fixed bin(31),
       2 Hook_exit_userword pointer,
       2 Hook_exit_ptr      pointer,
       2 Hook_exit_reserved pointer,
       2 Hook_exit_dsa      pointer,
       2 Hook_exit_addr     pointer,
       2 Hook_exit_end      char(0);

   Declare
     1 Exit_area       based(Hook_exit_ptr),
       2 Exit_bdata      pointer,
       2 Exit_pdata      pointer,
       2 Exit_epa        pointer,
       2 Exit_mod_end    pointer,
       2 Exit_a_visits   pointer,
       2 Exit_prev_mod   pointer,
       2 Exit_area_end   char(0);

   Declare (Addr,Entryaddr)  builtin;
   Declare (Stg,Sysnull)     builtin;

   Declare IBMBHKS external entry( fixed bin(31), fixed bin(31));

   Dcl HksFncStmt  fixed bin(31) init(1) static;
   Dcl HksFncEntry fixed bin(31) init(2) static;
   Dcl HksFncExit  fixed bin(31) init(3) static;
   Dcl HksFncPath  fixed bin(31) init(4) static;
   Dcl HksFncLabel fixed bin(31) init(5) static;
   Dcl HksFncBCall fixed bin(31) init(6) static;
   Dcl HksFncACall fixed bin(31) init(7) static;
   Dcl HksRetCode  fixed bin(31);
```

Figure 43 (Part 1 of 6). Sample facility 1: CEEBINT module

```
/*****************************************************************/
/*                                                               */
/* Following declares are used in setting up HOOKUP as the exit  */
/*                                                               */
/*****************************************************************/

Declare HOOKUP           external entry;

Declare IBMBSIR external entry( pointer );

Declare
  1 Sir_data,
                                /*                              */
    2 Sir_fnccode   fixed bin(31), /* Function code              */
                                /*   3: supply data for module */
                                /*   4: supply data for blocks */
                                /*   5: supply data for stmts  */
                                /*   6: supply data for paths  */
                                /*                              */
    2 Sir_retcode   fixed bin(31), /* Return code              */
                                /*   0: successful             */
                                /*   4: not compiled with      */
                                /*      appropriate TEST opt.  */
                                /*   8: not PL/I or not        */
                                /*      compiled with TEST     */
                                /*  12: unknown function code  */
                                /*                              */
    2 Sir_entry     entry,      /* Entry var for module         */
                                /*                              */
    2 Sir_mod_data  pointer,    /* A(module_data)               */
                                /*                              */
    2 Sir_a_data    pointer,    /* A(data for function code)    */
                                /*                              */
    2 Sir_end       char(0);    /*                              */

Declare
  1 Module_data,
                                /*                              */
    2 Module_len    fixed bin(31), /* = STG(Module_data)        */
                                /*                              */
    2 Module_options,           /* Compile time options         */
                                /*                              */
      3 Module_general_options, /*                              */
                                /*                              */
        4 Module_stmtno  BIT(01), /*   Stmt number table does   */
                                /*   not exist                  */
        4 Module_gonum   BIT(01), /*   Table has GONUMBER format */
        4 Module_cmpatv1 BIT(01), /*   compiled with CMPAT(V1)  */
        4 Module_graphic BIT(01), /*   compiled with GRAPHIC    */
        4 Module_opt     BIT(01), /*   compiled with OPTIMIZE   */
        4 Module_inter   BIT(01), /*   compiled with INTERRUPT  */
        4 Module_gen1x   BIT(02), /*   Reserved                 */
                                /*                              */
      3 Module_general_options2, /*                            */
                                /*                              */
        4 Module_gen2x   BIT(08), /*   Reserved                 */
                                /*                              */
      3 Module_test_options,    /*                              */
        4 Module_test    BIT(01), /*   compiled with TEST       */
        4 Module_stmt    BIT(01), /*   STMT  suboption is valid */
        4 Module_path    BIT(01), /*   PATH  suboption is valid */
        4 Module_block   BIT(01), /*   BLOCK suboption is valid */
        4 Module_testx   BIT(03), /*   Reserved                 */
        4 Module_sym     BIT(01), /*   SYM   suboption is valid */
                                /*                              */
```

Figure 43 (Part 2 of 6). Sample facility 1: CEEBINT module

```
      3 Module_sys_options,        /*                            */
        4 Module_cms      BIT(01), /*   SYSTEM(CMS)              */
        4 Module_cmstp    BIT(01), /*   SYSTEM(CMSTP)            */
        4 Module_mvs      BIT(01), /*   SYSTEM(MVS)              */
        4 Module_tso      BIT(01), /*   SYSTEM(TSO)              */
        4 Module_cics     BIT(01), /*   SYSTEM(CICS              */
        4 Module_ims      BIT(01), /*   SYSTEM(IMS)              */
        4 Module_vse      BIT(01), /*   SYSTEM(VSE)              */
        4 Module_sysx     BIT(01), /*   Reserved                 */
                                   /*                            */
    2 Module_blocks fixed bin(31), /* Count of blocks            */
                                   /*                            */
    2 Module_size   fixed bin(31), /* Size of module             */
                                   /*                            */
    2 Module_shooks fixed bin(31), /* Count of stmt hooks        */
                                   /*                            */
    2 Module_phooks fixed bin(31), /* Count of path hooks        */
                                   /*                            */
    2 Module_exts   fixed bin(31), /* Count of external entries  */
                                   /*                            */
    2 Module_data_end    char(0);
Declare
  1 Block_table       based(A_block_table),
    2 Block_a_data       pointer,
    2 Block_count        fixed bin(31),
    2 Block_data(Blocks refer(Block_count)),
      3 Block_offset     fixed bin(31),
      3 Block_size       fixed bin(31),
      3 Block_level      fixed bin(15),
      3 Block_parent     fixed bin(15),
      3 Block_child      fixed bin(15),
      3 Block_sibling    fixed bin(15),
      3 Block_name       char(34) varying,
    2 Block_table_end    char(0);

Declare
  1 Stmt_table        based(A_stmt_table),
    2 Stmt_a_data        pointer,
    2 Stmt_count         fixed bin(31),
    2 Stmt_data(Stmts  refer(Stmt_count)),
      3 Stmt_offset      fixed bin(31),
      3 Stmt_no          fixed bin(31),
      3 Stmt_lineno      fixed bin(15),
      3 Stmt_reserved    fixed bin(15),
      3 Stmt_type        fixed bin(15),
      3 Stmt_block       fixed bin(15),
    2 Stmt_table_end     char(0);

Declare
  1 Path_table        based(A_path_table),
    2 Path_a_data        pointer,
    2 Path_count         fixed bin(31),
    2 Path_data(Paths  refer(Path_count)),
      3 Path_offset      fixed bin(31),
      3 Path_no          fixed bin(31),
      3 Path_lineno      fixed bin(15),
      3 Path_reserved    fixed bin(15),
      3 Path_type        fixed bin(15),
      3 Path_block       fixed bin(15),
    2 Path_table_end     char(0);

Declare Blocks           fixed bin(31);
Declare Stmts            fixed bin(31);
Declare Paths            fixed bin(31);
Declare Visits           fixed bin(31);
```

Figure 43 (Part 3 of 6). Sample facility 1: CEEBINT module

```
Declare A_block_table     pointer;
Declare A_path_table      pointer;
Declare A_stmt_table      pointer;

Declare
  1 Hook_table        based(Exit_a_visits),
    2 Hook_data_size     fixed bin(31),
    2 Hook_data(Visits refer(Hook_data_size)),
      3 Hook_visits      fixed bin(31),
    2 Hook_data_end      char(0);

Declare A_visits          pointer;
Declare A_type            pointer;
Declare Previous_in_chain pointer;

/*********************************************************************/
/*                                                                   */
/* Following code is used to set up the hook exit control block      */
/*                                                                   */
/*********************************************************************/

Allocate Exit_list;
Exit_list_count = 1;

A_Exits = A_exit_list;

Allocate Hook_exit_block;
Hook_exit_len = Stg(Hook_Exit_block);

/*********************************************************************/
/*                                                                   */
/* Following code sets up HOOKUP as the hook exit                    */
/*                                                                   */
/*********************************************************************/

Hook_exit_rtn = Entryaddr(HOOKUP);

Previous_in_chain = Sysnull();

Call Add_module_to_list( A_main );

Call IBMBHKS( HksFncEntry, HksRetCode );
Call IBMBHKS( HksFncExit, HksRetCode );
Call IBMBHKS( HksFncPath, HksRetCode );
```

Figure 43 (Part 4 of 6). Sample facility 1: CEEBINT module

```
/*********************************************************************/
/*                                                                   */
/* Following subroutine retrieves all the static information         */
/* available on the MAIN routine and those linked with it            */
/*                                                                   */
/*********************************************************************/

Add_module_to_list: Proc( In_epa ) recursive;

  Dcl In_epa       pointer;
  Dcl Next_epa     pointer;
  Dcl Inx          fixed bin(31);

  Declare
    1 Exts_table        based(A_exts_table),
      2 Exts_a_data        pointer,
      2 Exts_count         fixed bin(31),
      2 Exts_data(Exts   refer(Exts_count)),
        3 Exts_epa         pointer,
      2 Exts_table_end     char(0);

  Declare Exts             fixed bin(31);
  Declare A_exts_table     pointer;

  Sir_fnccode = 3;

  Entryaddr(Sir_entry) = In_epa;
  Sir_mod_data = Addr(Module_data);

  Module_len = Stg(Module_data);

  Call IBMBSIR(Addr(Sir_data));

  If ( Sir_retcode = 0 )
   & ( Module_path ) then
    Do;
      Sir_fnccode = 4;

      Blocks = Module_blocks;
      Allocate Block_table;
      Block_a_data = ADDR(Block_data);
      Sir_a_data = A_block_table;

      Call IBMBSIR(Addr(Sir_data));

      Sir_fnccode = 6;

      Paths = Module_phooks;
      Allocate Path_table;
      Path_a_data = Addr(Path_data);
      Sir_a_data = A_path_table;

      Call IBMBSIR(Addr(Sir_data));

      /* Allocate areas needed                 */

      Allocate Exit_area;

      Exit_prev_mod = Previous_in_chain;
      Previous_in_chain = Hook_exit_ptr;

      Exit_pdata = A_path_table;
      Exit_bdata = A_block_table;

      Exit_epa = In_epa;
      Exit_mod_end = Exit_epa + module_size;
```

Figure 43 (Part 5 of 6). Sample facility 1: CEEBINT module

```
        Visits = Paths;
        Allocate Hook_table Set(Exit_a_visits);
        Hook_visits = 0;

        If Module_exts = 0 then;
        Else
          Do;
            Sir_fnccode = 7;

            Exts = Module_exts;
            Allocate Exts_table;
            Exts_a_data = Addr(Exts_data);
            Sir_a_data = A_exts_table;

            Call IBMBSIR(Addr(Sir_data));

            If Sir_retcode = 0 then
              Do;
                Do Inx = 1 to Exts;
                  Next_epa = A_exts_table->Exts_epa(Inx);
                  Call Add_module_to_list( Next_epa );
                End;
                Free Exts_table;
              End;
            Else;
          End;
      End;
    Else;

  End Add_module_to_list;

End;
```

Figure 43 (Part 6 of 6). Sample facility 1: CEEBINT module

.

```
*PROCESS FLAG(I) GOSTMT STMT SOURCE;
*PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
 HOOKUP: Proc( Exit_for_hooks ) reorder;

   Dcl Exit_for_hooks    pointer;    /* Address of exit list      */

   Declare
     1 Hook_exit_block   based(Exit_for_hooks),
       2 Hook_exit_len      fixed bin(31),
       2 Hook_exit_rtn      pointer,
       2 Hook_exit_fnccode  fixed bin(31),
       2 Hook_exit_retcode  fixed bin(31),
       2 Hook_exit_rsncode  fixed bin(31),
       2 Hook_exit_userword pointer,
       2 Hook_exit_ptr      pointer,
       2 Hook_exit_reserved pointer,
       2 Hook_exit_dsa      pointer,
       2 Hook_exit_addr     pointer,
       2 Hook_exit_end      char(0);

   Declare
     1 Exit_area       based(Module_data),
       2 Exit_bdata      pointer,
       2 Exit_pdata      pointer,
       2 Exit_epa        pointer,
       2 Exit_last       pointer,
       2 Exit_a_visits   pointer,
       2 Exit_prev_mod   pointer,
       2 Exit_area_end   char(0);

   Declare
     1 Path_table        based(Exit_pdata),
       2 Path_a_data        pointer,
       2 Path_count         fixed bin(31),
       2 Path_data(32767),
         3 Path_offset      fixed bin(31),
         3 Path_no          fixed bin(31),
         3 Path_lineno      fixed bin(15),
         3 Path_reserved    fixed bin(15),
         3 Path_type        fixed bin(15),
         3 Path_block       fixed bin(15),
       2 Path_table_end     char(0);


   Declare
     1 Block_table       based(Exit_bdata),
       2 Block_a_data       pointer,
       2 Block_count        fixed bin(31),
       2 Block_data(32767),
         3 Block_offset     fixed bin(31),
         3 Block_size       fixed bin(31),
         3 Block_level      fixed bin(15),
         3 Block_parent     fixed bin(15),
         3 Block_child      fixed bin(15),
         3 Block_sibling    fixed bin(15),
         3 Block_name       char(34) varying,
       2 Block_table_end    char(0);

   Declare
     1 Hook_table        based(Exit_a_visits),
       2 Hook_data_size     fixed bin(31),
       2 Hook_data(32767),
         3 Hook_visits      fixed bin(31),
       2 Hook_data_end      char(0);
```

Figure 44 (Part 1 of 4). Sample facility 1: HOOKUP program

```
Declare Ps              fixed bin(31);
Declare Ix              fixed bin(31);
Declare Jx              fixed bin(31);
Declare Total           float dec(06);
Declare Percent         Fixed dec(6,4);
Declare Col1            char(33);
Declare Col2            char(14);
Declare Col3            char(10);

Declare Sysnull         Builtin;

Declare Module_data     pointer;


Module_data = Hook_exit_ptr;

/*********************************************************************/
/*                                                                   */
/* Search for hook address in chain of modules                       */
/*                                                                   */
/*********************************************************************/

Do While ( Hook_exit_addr < Exit_epa | Hook_exit_addr > Exit_last )
   Until ( Module_data = Sysnull() );
  Module_data = Exit_prev_mod;
End;

/*********************************************************************/
/*                                                                   */
/* If not, found                                                     */
/*   IBMBHIR could be called to find address of entry point          */
/*   for the module                                                  */
/*   IBMBSIR could then be called as in CEEBINT to add               */
/*   module to the chain of known modules                            */
/*                                                                   */
/*********************************************************************/

If Module_data = Sysnull() then;
Else
  Do;
    Ps = Hook_exit_addr - Exit_epa;

    /*****************************************************************/
    /*                                                               */
    /*  A binary search could be done here and such a search         */
    /*  would be much more efficient for large programs              */
    /*                                                               */
    /*****************************************************************/

    Do Ix = 1 to Path_count
      While ( Ps ¬= Path_offset(Ix) );
    End;

    If (Ix > 0) & (Ix <= Path_count) then
      Hook_visits(Ix) = Hook_visits(Ix) + 1;
    Else;
```

Figure 44 (Part 2 of 4). Sample facility 1: HOOKUP program

```
      /*****************************************************************/
      /*                                                               */
      /*  If hook type is for block exit                               */
      /*   AND                                                         */
      /*     block being exited is the first block in a procedure      */
      /*   AND                                                         */
      /*     that procedure is the MAIN procedure, then                */
      /*   invoke the post processing routine                          */
      /*                                                               */
      /*  Note that these conditions might never be met, for           */
      /*  example, if SIGNAL FINISH were issued or if an               */
      /*  EXEC CICS RETURN were issued                                 */
      /*                                                               */
      /*****************************************************************/

      If Path_type(Ix) = 2
       & Path_block(Ix) = 1
       & Exit_prev_mod = Sysnull() then
        Do;
          Put skip list ( ' ' );
          Put skip list ( ' ' );

          Put skip list ( 'Post processing' );

          Module_data = Hook_exit_ptr;

          Do Until ( Module_data = Sysnull() );
            Call Report_data;
            Module_data = Exit_prev_mod;
          End;

        End;
      Else;
    End;

  Hook_exit_retcode = 4;
  Hook_exit_rsncode = 0;


  Report_data: Proc;

    Total = 0;
    Do Jx = 1 to Path_count;
      Total = Total + Hook_visits(Jx);
    End;

    Put skip list ( ' ' );
    Do Jx = 1 to Path_count;
      If Path_type(Jx) = 1 then
        do;
          Put skip list ( ' ' );
          Put skip list ( 'Data for block ' ||
                 Block_name(Path_block(Jx)) );
          Put skip list ( ' ' );
          Col1 = '     Statement  Type';
          Col2 = '        Visits';
          Col3 = '   Percent';
          Put skip list ( Col1 || ' ' || Col2 || ' ' || Col3 );
          Put skip list ( ' ' );
        end;
      Else;
```

Figure 44 (Part 3 of 4). Sample facility 1: HOOKUP program

```
      Select ( Path_type(Jx) );
        When ( 1 )
          Col1 = Path_no(Jx) || '  block entry';
        When ( 2 )
          Col1 = Path_no(Jx) || '  block exit';
        When ( 3 )
          Col1 = Path_no(Jx) || '  label';
        When ( 4 )
          Col1 = Path_no(Jx) || '  before call';
        When ( 5 )
          Col1 = Path_no(Jx) || '  after call';
        When ( 6 )
          Col1 = Path_no(Jx) || '  start of do loop';
        When ( 7 )
          Col1 = Path_no(Jx) || '  if-true';
        When ( 8 )
          Col1 = Path_no(Jx) || '  if-false';
        Otherwise
          Col1 = Path_no(Jx);
      End;

      Col2 = Hook_visits(Jx);
      Percent = 100 * (Hook_visits(Jx)/Total);
      Put skip list ( Col1 || ' ' || Col2 || '  ' || Percent );
    End;

    Put skip list ( ' ' );
    Put skip list ( ' ' );
    Put skip list ( ' ' );

  End Report_data;

End;
```

Figure 44 (Part 4 of 4). Sample facility 1: HOOKUP program

Sample facility 2: Performing function tracing

The following example programs show how to establish a rather simple hook exit to perform function tracing.

Overall setup

This facility consists of two programs: CEEBINT and HOOKUPT. The CEEBINT module is coded such that:

- A hook exit to the HOOKUPT program is established
- Calls to IBMBHKS are made to set hooks prior to execution.

At run time, whenever HOOKUPT gains control (via the established hook exit), it calls IBMBHIR to obtain information to create a function trace.

Output generated

The output given in Figure 45 below is generated during execution of a PL/I program called KNIGHT. The KNIGHT program's function is to determine the moves a knight must make to land on each square of a chess board only once.

The output is created by the HOOKUPT program as employed in this facility.

```
Entry hook in KNIGHT
Exit hook in KNIGHT
```

Figure 45. Function trace produced by sample facility 2

Note: In a more complicated program, many more entry and exit messages would be produced.

Source code

The source code for Sample Facility 2 is distributed with the PL/I VSE product, and is shown here for easy reference. The sample routines are:

Routine name	Distributed as	Shown in
CEEBINT	IELS2INT.P	Figure 46 on page 201
HOOKUPT	IELS2HKP.P	Figure 47 on page 203

```
*PROCESS FLAG(I) GOSTMT STMT SOURCE;
*PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
 CEEBINT: Proc( Number, RetCode, RsnCode, FncCode, A_Main,
                UserWd, A_Exits )
                options(reentrant) reorder;

   Dcl Number     fixed bin(31);      /* Number of args = 7        */
   Dcl RetCode    fixed bin(31);      /* Return Code = 0           */
   Dcl RsnCode    fixed bin(31);      /* Reason Code = 0           */
   Dcl FncCode    fixed bin(31);      /* Function Code = 1         */
   Dcl A_Main     pointer;            /* Address of Main Routine   */
   Dcl UserWd     fixed bin(31);      /* User Word                 */
   Dcl A_Exits    pointer;            /* A(Exits list)             */

   Declare A_exit_list       pointer;

   Declare
     1 Exit_list         based(A_exit_list),
       2 Exit_list_count    fixed bin(31),
       2 Exit_list_slots,
         3 Exit_for_hooks   pointer,
       2 Exit_list_end      char(0);

   Declare
     1 Hook_exit_block   based(Exit_for_hooks),
       2 Hook_exit_len      fixed bin(31),
       2 Hook_exit_rtn      pointer,
       2 Hook_exit_fnccode  fixed bin(31),
       2 Hook_exit_retcode  fixed bin(31),
       2 Hook_exit_rsncode  fixed bin(31),
       2 Hook_exit_userword pointer,
       2 Hook_exit_ptr      pointer,
       2 Hook_exit_reserved pointer,
       2 Hook_exit_dsa      pointer,
       2 Hook_exit_addr     pointer,
       2 Hook_exit_end      char(0);

   Declare
     1 Exit_area       based(Hook_exit_ptr),
       2 Exit_bdata      pointer,
       2 Exit_pdata      pointer,
       2 Exit_epa        pointer,
       2 Exit_mod_end    pointer,
       2 Exit_a_visits   pointer,
       2 Exit_prev_mod   pointer,
       2 Exit_area_end   char(0);

   Declare (Addr,Entryaddr)  builtin;
   Declare (Stg,Sysnull)     builtin;

   Declare IBMBHKS external entry( fixed bin(31), fixed bin(31));

   Dcl HksFncStmt  fixed bin(31) init(1) static;
   Dcl HksFncEntry fixed bin(31) init(2) static;
   Dcl HksFncExit  fixed bin(31) init(3) static;
   Dcl HksFncPath  fixed bin(31) init(4) static;
   Dcl HksFncLabel fixed bin(31) init(5) static;
   Dcl HksFncBCall fixed bin(31) init(6) static;
   Dcl HksFncACall fixed bin(31) init(7) static;
   Dcl HksRetCode  fixed bin(31);
```

Figure 46 (Part 1 of 2). Sample facility 2: CEEBINT module

```
  /*******************************************************************/
  /*                                                                 */
  /* Following code is used to set up the hook exit control block    */
  /*                                                                 */
  /*******************************************************************/

  Allocate Exit_list;
  Exit_list_count = 1;

  A_Exits = A_exit_list;

  Allocate Hook_exit_block;
  Hook_exit_len = Stg(Hook_Exit_block);

  /*******************************************************************/
  /*                                                                 */
  /* Following code sets up HOOKUPT as the hook exit                 */
  /*                                                                 */
  /*******************************************************************/

  Declare HOOKUPT   external entry;

  Hook_exit_rtn = Entryaddr(HOOKUPT);

  Call IBMBHKS( HksFncEntry, HksRetCode );
  Call IBMBHKS( HksFncExit, HksRetCode );

End;
```

Figure 46 (Part 2 of 2). Sample facility 2: CEEBINT module

```
*PROCESS FLAG(I) GOSTMT STMT SOURCE;
*PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
 HOOKUPT: Proc( Exit_for_hooks ) reorder;

   Dcl Exit_for_hooks    pointer;    /* Address of exit list      */

   Declare
     1 Hook_exit_block   based(Exit_for_hooks),
       2 Hook_exit_len      fixed bin(31),
       2 Hook_exit_rtn      pointer,
       2 Hook_exit_fnccode  fixed bin(31),
       2 Hook_exit_retcode  fixed bin(31),
       2 Hook_exit_rsncode  fixed bin(31),
       2 Hook_exit_userword pointer,
       2 Hook_exit_ptr      pointer,
       2 Hook_exit_reserved pointer,
       2 Hook_exit_dsa      pointer,
       2 Hook_exit_addr     pointer,
       2 Hook_exit_end      char(0);

   Declare
     1 Hook_data,
       2 Hook_stg           fixed bin(31),
       2 Hook_epa           pointer,
       2 Hook_lang_code     aligned bit(8),
       2 Hook_path_code     aligned bit(8),
       2 Hook_name_len      fixed bin(15),
       2 Hook_name_addr     pointer,
       2 Hook_block_count   fixed bin(31),
       2 Hook_reserved      fixed bin(31),
       2 Hook_data_end      char(0);

   Declare IBMBHIR          external entry;
   Declare Chars            char(256) based;
   Declare (Addr,Substr)    builtin;


   Call IBMBHIR( Addr(Hook_data), Hook_exit_dsa, Hook_exit_addr );

   If Hook_block_count = 1 then
     Select ( Hook_path_code );
       When ( 1 )
         Put skip list( 'Entry hook in ' ||
                        Substr(Hook_name_addr->Chars,1,Hook_name_len) );
       When ( 2 )
         Put skip list( 'Exit hook in ' ||
                        Substr(Hook_name_addr->Chars,1,Hook_name_len) );
       Otherwise
         ;
     End;
   Else;

 End;
```

Figure 47. Sample facility 2: HOOKUPT program

Sample facility 3: Analyzing CPU-time usage

This facility extends the code-coverage function of Sample Facility 1 to also report on CPU-time usage.

Overall setup

This facility consists of four programs: CEEBINT, HOOKUP, TIMINI, and TIMCPU. The CEEBINT module is coded so that:

- A hook exit to the HOOKUP program is established
- Calls to IBMBHKS are made to set hooks prior to execution.

At run time, whenever HOOKUP gains control (via the established hook exit), it calls IBMBSIR to obtain code coverage information on the MAIN procedure and those linked with it. This is identical to the function of Sample Facility 1.

In addition, this HOOKUP program makes calls to the assembler routines TIMINI and TIMCPU to obtain information on CPU-time usage.

The SPROG suboption of the LANGLVL compile-time option is specified in order to enable the adding to a pointer that takes place in CEEBINT and the comparing of two pointers that takes place in HOOKUP.

The TIMINI and TIMCPU assembler subroutines use the SETT and TESTT macro instructions. These instructions place restrictions on the running of the program, as follows:

- The program must be link-edited with AMODE(24) and RMODE(24).
- The LE/VSE run-time option HEAP(,,BELOW) must be used.
- The program must run in the partition for which task timer support has been generated in the VSE/ESA supervisor (TTIME=*partition* in the FOPT macro instruction).

Output generated

The output given in Figure 48 on page 205 is generated during execution of a sample program named EXP98. The main procedure was compiled with the TEST(ALL) option.

The output is created by the HOOKUP program as employed in this facility.

Data for block EXP98

Statement Type	--- Visits --- Number	Percent	--- CPU Time --- Milliseconds	Percent
1 block entry	1	0.0201		
16 start of do loop	24	0.4832	83.768	0.6036
26 start of do loop	38	0.7652	107.254	0.7729
27 start of do loop	38	0.7652	102.159	0.7362
28 if true	38	0.7652	103.066	0.7427
30 start of do loop	456	9.1824	1233.827	8.8915
31 before call	456	9.1824	1237.831	8.9203
31 after call	456	9.1824	1238.400	8.9244
41 before call	38	0.7652	102.491	0.7385
41 after call	38	0.7652	103.038	0.7425
49 start of do loop	38	0.7652	102.029	0.7352
50 if true	0	0.0000	0.000	0.0000
51 if true	0	0.0000	0.000	0.0000
52 if true	0	0.0000	0.000	0.0000
54 start of do loop	304	6.1216	827.971	5.9667
55 if true	76	1.5304	206.831	1.4905
56 if true	0	0.0000	0.000	0.0000
57 if true	38	0.7652	104.351	0.7520
61 block exit	1	0.0201	2.703	0.0194
Totals for block	2040	41.0790	5555.719	40.0364

Data for block V1B

Statement Type	--- Visits --- Number	Percent	--- CPU Time --- Milliseconds	Percent
32 block entry	456	9.1824		
33 start of do loop	456	9.1824	1251.487	9.0187
34 if true	456	9.1824	1251.125	9.0161
35 if true	456	9.1824	1501.528	10.8206
36 if true	0	0.0000	0.000	0.0000
37 if true	456	9.1824	1275.072	9.1887
39 block exit	456	9.1824	1256.781	9.0569
Totals for block	2736	55.0944	6535.993	47.1010

Data for block V1C

Statement Type	--- Visits --- Number	Percent	--- CPU Time --- Milliseconds	Percent
42 block entry	38	0.7652		
43 start of do loop	38	0.7652	105.529	0.7604
44 if true	38	0.7652	106.233	0.7655
45 if true	0	0.0000	0.000	0.0000
46 if true	38	0.7652	106.481	0.7673
48 block exit	38	0.7652	104.547	0.7534
Totals for block	190	3.8260	422.790	3.0466

Figure 48. CPU-time usage and code coverage reported by sample facility 3

The performance of this facility depends on the number of hook exits invoked. Collecting the data above increased the CPU time of EXP98 by approximately forty-four times. Each CPU-time measurement indicates the amount of virtual CPU time that was used since the previous hook was executed. Note that the previous hook is not necessarily the previous hook in the figure.

In the above data, the percent columns for the number of visits and the CPU time are very similar. This will not always be the case, especially where hidden code (like library calls or supervisor services) is involved.

Source code

The source code for Sample Facility 3 is distributed with the PL/I VSE product, and is shown here for easy reference. The sample routines are:

Routine name	Distributed as	Shown in
CEEBINT	IELS3INT.P	Figure 49
HOOKUP	IELS3HKP.P	Figure 50 on page 213
TIMINI	IELS3TIN.A	Figure 51 on page 218
TIMCPU	IELS3TCP.A	Figure 52 on page 219

```
*PROCESS TEST(NONE,NOSYM) LANGLVL(SPROG);
 CEEBINT: PROC( Number, RetCode, RsnCode, FncCode, A_Main,
                UserWd, A_Exits )
                OPTIONS(REENTRANT) REORDER;

   DCL Number     FIXED BIN(31);      /* Number of args = 7        */
   DCL RetCode    FIXED BIN(31);      /* Return Code = 0           */
   DCL RsnCode    FIXED BIN(31);      /* Reason Code = 0           */
   DCL FncCode    FIXED BIN(31);      /* Function Code = 1         */
   DCL A_Main     POINTER;            /* Address of Main Routine   */
   DCL UserWd     FIXED BIN(31);      /* User Word                 */
   DCL A_Exits    POINTER;            /* A(Exits list)             */

   /*****************************************************************/
   /* This routine gets invoked at initialization of the MAIN       */
   /* Structures and variables for use in establishing a hook exit  */
   /*                                                               */
   /*****************************************************************/

   DECLARE A_exit_list       POINTER;
   DECLARE Based_ptr         POINTER BASED;
   DECLARE Entry_var         ENTRY VARIABLE;

   DECLARE Entryaddr         BUILTIN;
   DECLARE (Stg,Sysnull)     BUILTIN;

   DECLARE
     1 Exit_list         BASED(A_exit_list),
       2 Exit_list_count    FIXED BIN(31),
       2 Exit_list_slots,
         3 Exit_for_hooks   POINTER,
       2 Exit_list_end      CHAR(0);

   DECLARE
     1 Hook_exit_block   BASED(Exit_for_hooks),
       2 Hook_exit_len      FIXED BIN(31),
       2 Hook_exit_rtn      POINTER,
       2 Hook_exit_fnccode  FIXED BIN(31),
       2 Hook_exit_retcode  FIXED BIN(31),
       2 Hook_exit_rsncode  FIXED BIN(31),
       2 Hook_exit_userword FIXED BIN(31),
       2 Hook_exit_ptr      POINTER,
       2 Hook_exit_reserved POINTER,
       2 Hook_exit_dsa      POINTER,
       2 Hook_exit_addr     POINTER,
       2 Hook_exit_end      CHAR(0);

   DECLARE
     1 Exit_area         BASED(Hook_exit_ptr),
       2 Exit_bdata         POINTER,
       2 Exit_pdata         POINTER,
       2 Exit_epa           POINTER,
       2 Exit_xtra          POINTER,
       2 Exit_area_end      CHAR(0);
```

Figure 49 (Part 1 of 7). Sample facility 3: CEEBINT module

```
DECLARE
  1 Hook_table        BASED(Exit_xtra),
    2 Hook_data_size     FIXED BIN(31),
    2 Hook_data(Visits REFER(Hook_data_size)),
      3 Hook_visits      FIXED BIN(31),
    2 Hook_data_end      CHAR(0);

   /* the name of the routine that gets control at hook exit */

   DECLARE HOOKUP           EXTERNAL ENTRY ( POINTER );

/*******************************************************************/
/*                                                                 */
/* End of structures and variables for use with hook exit          */
/*                                                                 */
/*******************************************************************/

/*******************************************************************/
/*                                                                 */
/* Structures and variables for use in invoking IBMBSIR            */
/*                                                                 */
/*******************************************************************/

DECLARE
  1 Sir_data,
                                /*                                 */
    2 Sir_fnccode   FIXED BIN(31), /* Function code                */
                                /*   1: supply data for module     */
                                /*   2: supply data for blocks     */
                                /*   3: supply data for stmts      */
                                /*   4: supply data for paths      */
                                /*                                 */
    2 Sir_retcode   FIXED BIN(31), /* Return code                  */
                                /*   0: successful                 */
                                /*   4: not compiled with          */
                                /*      appropriate TEST opt.      */
                                /*   8: not PL/I or not            */
                                /*      compiled with TEST         */
                                /*  12: unknown function code      */
                                /*                                 */
    2 Sir_entry     ENTRY,      /* Entry var for module            */
                                /*                                 */
    2 Sir_mod_data  POINTER,    /* A(module_data)                  */
                                /*                                 */
    2 Sir_a_data    POINTER,    /* A(data for function code)       */
                                /*                                 */
    2 Sir_end       CHAR(0);    /*                                 */
                                /*                                 */

DECLARE
  1 Module_data,
                                /*                                 */
    2 Module_len    FIXED BIN(31), /* = STG(Module_data)           */
                                /*                                 */
    2 Module_options,           /* Compile time options            */
                                /*                                 */
      3 Module_general_options, /*                                 */
                                /*                                 */
        4 Module_stmtno  BIT(01), /*   Stmt number table does      */
                                /*   not exist                     */
```

Figure 49 (Part 2 of 7). Sample facility 3: CEEBINT module

```
        4 Module_gonum   BIT(01),  /*   Table has GONUMBER format */
        4 Module_cmpatv1 BIT(01),  /*   compiled with CMPAT(V1)   */
        4 Module_graphic BIT(01),  /*   compiled with GRAPHIC     */
        4 Module_opt     BIT(01),  /*   compiled with OPTIMIZE    */
        4 Module_inter   BIT(01),  /*   compiled with INTERRUPT   */
        4 Module_gen1x   BIT(02),  /*   Reserved                  */
                                   /*                             */
      3 Module_general_options2,   /*                             */
                                   /*                             */
        4 Module_gen2x   BIT(08),  /*   Reserved                  */
                                   /*                             */
      3 Module_test_options,       /*                             */
        4 Module_test    BIT(01),  /*   compiled with TEST        */
        4 Module_stmt    BIT(01),  /*   STMT  suboption is valid  */
        4 Module_path    BIT(01),  /*   PATH  suboption is valid  */
        4 Module_block   BIT(01),  /*   BLOCK suboption is valid  */
        4 Module_testx   BIT(03),  /*   Reserved                  */
        4 Module_sym     BIT(01),  /*   SYM   suboption is valid  */
                                   /*                             */
      3 Module_sys_options,        /*                             */
        4 Module_cms     BIT(01),  /*   SYSTEM(CMS)               */
        4 Module_cmstp   BIT(01),  /*   SYSTEM(CMSTP)             */
        4 Module_mvs     BIT(01),  /*   SYSTEM(MVS)               */
        4 Module_tso     BIT(01),  /*   SYSTEM(TSO)               */
        4 Module_cics    BIT(01),  /*   SYSTEM(CICS               */
        4 Module_ims     BIT(01),  /*   SYSTEM(IMS)               */
        4 Module_vse     BIT(01),  /*   SYSTEM(VSE)               */
        4 Module_sysx    BIT(01),  /*   Reserved                  */

    2 Module_blocks FIXED BIN(31), /* Count of blocks             */
                                   /*                             */
    2 Module_size   FIXED BIN(31), /* Size of module              */
                                   /*                             */
    2 Module_shooks FIXED BIN(31), /* Count of stmt hooks         */
                                   /*                             */
    2 Module_phooks FIXED BIN(31), /* Count of path hooks         */
                                   /*                             */
    2 Module_data_end    CHAR(0);

DECLARE
  1 Block_table       BASED(A_block_table),
    2 Block_a_data       POINTER,
    2 Block_count        FIXED BIN(31),
    2 Block_data(Blocks REFER(Block_count)),
      3 Block_offset     FIXED BIN(31),
      3 Block_size       FIXED BIN(31),
      3 Block_level      FIXED BIN(15),
      3 Block_parent     FIXED BIN(15),
      3 Block_child      FIXED BIN(15),
      3 Block_sibling    FIXED BIN(15),
      3 Block_name       CHAR(34) VARYING,
    2 Block_table_end    CHAR(0);

DECLARE
  1 Stmt_table        BASED(A_stmt_table),
    2 Stmt_a_data        POINTER,
    2 Stmt_count         FIXED BIN(31),
    2 Stmt_data(Stmts  REFER(Stmt_count)),
      3 Stmt_offset      FIXED BIN(31),
      3 Stmt_no          FIXED BIN(31),
      3 Stmt_lineno      FIXED BIN(15),
      3 Stmt_reserved    FIXED BIN(15),
      3 Stmt_type        FIXED BIN(15),
      3 Stmt_block       FIXED BIN(15),
    2 Stmt_table_end     CHAR(0);
```

Figure 49 (Part 3 of 7). Sample facility 3: CEEBINT module

```
DECLARE
  1 Path_table        BASED(A_path_table),
    2 Path_a_data        POINTER,
    2 Path_count         FIXED BIN(31),
    2 Path_data(Paths  REFER(Path_count)),
      3 Path_offset      FIXED BIN(31),
      3 Path_no          FIXED BIN(31),
      3 Path_lineno      FIXED BIN(15),
      3 Path_reserved    FIXED BIN(15),
      3 Path_type        FIXED BIN(15),
      3 Path_block       FIXED BIN(15),
    2 Path_table_end     CHAR(0);

DECLARE Blocks           FIXED BIN(31);
DECLARE Stmts            FIXED BIN(31);
DECLARE Paths            FIXED BIN(31);
DECLARE Visits           FIXED BIN(31);

DECLARE A_block_table    POINTER;
DECLARE A_stmt_table     POINTER;
DECLARE A_path_table     POINTER;

DECLARE Addr              BUILTIN;


Declare IBMBHKS external entry( fixed bin(31), fixed bin(31));

Dcl HksFncStmt  fixed bin(31) init(1) static;
Dcl HksFncEntry fixed bin(31) init(2) static;
Dcl HksFncExit  fixed bin(31) init(3) static;
Dcl HksFncPath  fixed bin(31) init(4) static;
Dcl HksFncLabel fixed bin(31) init(5) static;
Dcl HksFncBCall fixed bin(31) init(6) static;
Dcl HksFncACall fixed bin(31) init(7) static;
Dcl HksRetCode  fixed bin(31);

DECLARE IBMBSIR          EXTERNAL ENTRY ( POINTER );

/*******************************************************************/
/*                                                                 */
/* End of structures and variables for use in invoking IBMBSIR     */
/*                                                                 */
/*******************************************************************/
```

Figure 49 (Part 4 of 7). Sample facility 3: CEEBINT module

```
/*****************************************************************/
/*                                                               */
/* Sample code for invoking IBMBSIR                              */
/*                                                               */
/* IBMBSIR is first invoked to get the block and hook counts.    */
/*                                                               */
/* If that invocation is successful, the block table is allocated */
/* and IBMBSIR is invoked to fill in that table.                 */
/*                                                               */
/* If that invocation is also successful, the path hook table is */
/* allocated and IBMBSIR is invoked to fill in that table.       */
/*                                                               */
/*****************************************************************/


Sir_fnccode = 3;

/*- If counts are to be obtained from a non-MAIN routine   -*/
/*- then put comments around the "Entryaddr(Sir_entry) =   -*/
/*- A_Main" statement and remove the comments from         -*/
/*- around the DCL and the "Sir_entry = P00" statements    -*/
/*- and change P00 (both places) to the name of the        -*/
/*- non-MAIN routine:                                      -*/

      Entryaddr(Sir_entry) = A_main;
/*-   DCL P00 External entry;                              -*/
/*-   Sir_entry = P00;                                     -*/

Sir_mod_data = Addr(Module_data);

Call IBMBSIR( Addr(Sir_data) );

If Sir_retcode = 0 then
  Do;
    Sir_fnccode = 4;

    Blocks = Module_blocks;
    Allocate Block_table;
    Block_a_data = Addr(Block_data);
    Sir_a_data = A_block_table;

    Call IBMBSIR( Addr(Sir_data) );

    If Sir_retcode = 0 then
      Do;
        Sir_fnccode = 6;

        Paths = Module_phooks;
        Allocate Path_table;
        Path_a_data = Addr(Path_data);
        Sir_a_data = A_path_table;

        Call IBMBSIR( Addr(Sir_data) );
      End;
    Else
      Do ;
        Put skip list('CEEBINT MSG_2: Sir_retcode was not zero');
        Put skip list('CEEBINT MSG_2: Sir_retcode = ',Sir_retcode);
        Put skip list('CEEBINT MSG_2: Path table not allocated');
      End;
  End;
```

Figure 49 (Part 5 of 7). Sample facility 3: CEEBINT module

```
Else
  Do ;
    Put skip list('CEEBINT MSG_1: Sir_retcode was not zero');
    Put skip list('CEEBINT MSG_1: Sir_retcode = ',Sir_retcode);
    Put skip list('CEEBINT MSG_1: Block table not allocated');
    Put skip list('CEEBINT MSG_1: Path table not allocated');
  End;

/*********************************************************************/
/*                                                                   */
/* End of sample code for invoking IBMBSIR                           */
/*                                                                   */
/*********************************************************************/

/*********************************************************************/
/*                                                                   */
/* Sample code for setting up hook exit                              */
/*                                                                   */
/* The first two sets of instructions merely create an exit list  */
/* containing one element and create the hook exit block to which */
/* that element will point. Note that the hook exit is enabled    */
/* by this code, but it is not activated since Hook_exit_rtn = 0. */
/*                                                                   */
/*********************************************************************/


Allocate Exit_list;
A_Exits = A_exit_list;
Exit_list_count = 1;

Allocate Hook_exit_block;
Hook_exit_len = Stg(Hook_Exit_block);
Hook_exit_userword = UserWd;
Hook_exit_rtn = Sysnull();
Hook_exit_ptr = Sysnull();


/*********************************************************************/
/*                                                                   */
/*                                                                   */
/* The following code will cause the hook exit to invoke a        */
/* routine called HOOKUP that will keep count of how often each   */
/* path hook in the MAIN routine is executed.                     */
/*                                                                   */
/* First, the address of HOOKUP is put into the slot for the      */
/* address of the routine to be invoked when each hook is hit.    */
/*                                                                   */
/* Then, pointers to the block table and the path hook table      */
/* obtained from IBMBSIR in the sample code above are put into    */
/* the exit area. Next, the address of the routine being          */
/* monitored, in this case the MAIN routine, is put into the      */
/* exit area. Additionally, a table to keep count of the number   */
/* of visits is created, and its address is also put into the     */
/* exit area.                                                        */
/*                                                                   */
/* Finally IBMBHKS is invoked to turn on the PATH hooks.          */
/*                                                                   */
/*********************************************************************/

If Sir_retcode = 0 then
  Do;
    Entry_var = HOOKUP;
    Hook_exit_rtn = Addr(Entry_var)->Based_ptr;

    Allocate Exit_area;
```

Figure 49 (Part 6 of 7). Sample facility 3: CEEBINT module

```
      Exit_bdata = A_block_table;
      Exit_pdata = A_path_table;

      /*- If counts are to be obtained from a non-MAIN    -*/
      /*- routine then replace "A_Main" in the following  -*/
      /*- statement with the name of the non-MAIN routine -*/
      /*- (the name declared earlier as EXTERNAL ENTRY).  -*/

      Exit_epa = A_main;

      Visits = Paths;
      Allocate Hook_table;
      Hook_visits = 0;

      Call IBMBHKS( HksFncPath, HksRetCode );
    End;
  Else
    Do ;
      Put skip list('CEEBINT MSG_1: Sir_retcode was not zero');
      Put skip list('CEEBINT MSG_1: Sir_retcode = ',Sir_retcode);
      Put skip list('CEEBINT MSG_1: Exit area not allocated');
      Put skip list('CEEBINT MSG_1: Hook table not allocated');
    End;


  /*********************************************************************/
  /*                                                                   */
  /* End of sample code for setting up hook exit                       */
  /*                                                                   */
  /*********************************************************************/



  /*********************************************************************/
  /*                                                                   */
  /* Place your actual user code here                                  */
  /*                                                                   */
  /*********************************************************************/
  Put skip list('CEEBINT: At end of CEEBINT initialization');

END CEEBINT;
```

Figure 49 (Part 7 of 7). Sample facility 3: CEEBINT module

```
*PROCESS TEST(NONE,NOSYM) LANGLVL(SPROG);
 Hookup : PROC( exit_for_hooks );

   DECLARE  Exit_for_hooks  Pointer;

   DECLARE
     1 Hook_exit_block   BASED(Exit_for_hooks),
       2 Hook_exit_len      FIXED BIN(31),
       2 Hook_exit_rtn      POINTER,
       2 Hook_exit_fnccode  FIXED BIN(31),
       2 Hook_exit_retcode  FIXED BIN(31),
       2 Hook_exit_rsncode  FIXED BIN(31),
       2 Hook_exit_userword POINTER,
       2 Hook_exit_ptr      POINTER,
       2 Hook_exit_reserved POINTER,
       2 Hook_exit_dsa      POINTER,
       2 Hook_exit_addr     POINTER,
       2 Hook_exit_end      CHAR(0);

   DECLARE
     1 Exit_area         BASED(Hook_exit_ptr),
       2 Exit_bdata         POINTER,
       2 Exit_pdata         POINTER,
       2 Exit_epa           POINTER,
       2 Exit_a_visits      POINTER,
       2 Exit_area_end      CHAR(0);


   DECLARE
     1 Block_table       BASED(exit_bdata),
       2 Block_a_data       POINTER,
       2 Block_count        FIXED BIN(31),
       2 Block_data(Blocks REFER(Block_count)),
         3 Block_offset     FIXED BIN(31),
         3 Block_size       FIXED BIN(31),
         3 Block_level      FIXED BIN(15),
         3 Block_parent     FIXED BIN(15),
         3 Block_child      FIXED BIN(15),
         3 Block_sibling    FIXED BIN(15),
         3 Block_name       CHAR(34) VARYING,
       2 Block_table_end    CHAR(0);


   DECLARE
     1 Path_table        BASED(exit_pdata),
       2 Path_a_data        POINTER,
       2 Path_count         FIXED BIN(31),
       2 Path_data(Paths  REFER(Path_count)),
         3 Path_offset      FIXED BIN(31),
         3 Path_no          FIXED BIN(31),
         3 Path_lineno      FIXED BIN(15),
         3 Path_reserved    FIXED BIN(15),
         3 Path_type        FIXED BIN(15),
         3 Path_block       FIXED BIN(15),
       2 Path_table_end     CHAR(0);

   DECLARE
     1 Hook_table        BASED(Exit_a_visits),
       2 Hook_data_size     FIXED BIN(31),
       2 Hook_data(Visits REFER(Hook_data_size)),
         3 Hook_visits      FIXED BIN(31),
       2 Hook_data_end      CHAR(0);


   DECLARE Ps               FIXED BIN(31);
   DECLARE Ix               FIXED BIN(31);
   DECLARE Jx               FIXED BIN(31);
```

Figure 50 (Part 1 of 5). Sample facility 3: HOOKUP program

```
/* ---------- Notes on cpu timing ---------------------- */
/*                                                       */
/*  TIMCPU is an Assembler routine which uses the TESTT  */
/*  macro to get the task CPU time, and converts it      */
/*  to microseconds.                                     */
/*  The TIMINI routine initializes the timer using the   */
/*  SETT macro. The timer has a maximum value of one     */
/*  thousand seconds.                                    */

/* -------------- Declares for cpu timing -------------- */
DCL  TIMINI              ENTRY OPTIONS (ASSEMBLER);
DCL  TIMCPU              ENTRY OPTIONS (ASSEMBLER);
DCL  Decimal             BUILTIN;
DCL  cpu_visits          FIXED BIN(31) STATIC;  /* = visits */
DCL  Hook_cpu_visits     Pointer STATIC  ;
DCL  exe_mon     FIXED BIN(31) EXTERNAL;
  /* exe_mon <=0  no cpu timing data is printed          */
  /* exe_mon  =1  only block data is obtained - default  */
  /* exe_mon >=2  all hooks are timed                    */
DECLARE
  1 Hook_time_table        BASED(Hook_cpu_visits),
   2 Hooksw,
       3 BLK_entry_time       FIXED BIN(31,0),
       3 Before               FIXED BIN(31,0),
       3 After                FIXED BIN(31,0),
       3 Temp_cpu             FIXED BIN(31,0),
    2 Hook_time_data_size     FIXED BIN(31),
    2 Hook_time_data(cpu_visits Refer(Hook_time_data_size)),
      3 Hook_time             Fixed Bin(31),
    2 Hook_time_data_end      Char(0);
/* ---------- END Declares for cpu timing -------------- */

/* compute offset */
ps = hook_exit_addr - exit_epa;

/* search for hook */
do ix = 1 to path_count
   while ( ps ¬= path_offset(ix) );
end;

/* if hook found - update table */
if (ix > 0) & (ix <=path_count) then do;
   hook_visits(ix) = hook_visits(ix) + 1;

   /* -------- this SELECT gets the cpu timings --------  */
   Select;
     When(path_type(ix)=1 & ix=1) do;  /* external block entry */
       exe_mon = 1;  /* default can be overridden in Main */
       cpu_visits = hook_data_size;  /* number of hooks */
       allocate hook_time_table;  /* cpu time table */
       hook_time = 0;  /* initialize the table */
       call TIMINI;               /* initialize timers */
       call TIMCPU  (BLK_entry_time);  /* get block entry time */
       hooksw.before = BLK_entry_time;      /* block entry */
       hooksw.after  = BLK_entry_time;      /* block entry */
     end;  /* end of the When(1) do */
```

Figure 50 (Part 2 of 5). Sample facility 3: HOOKUP program

```
      Otherwise do;
        if (exe_mon > 0) then do;
          /* collect cpu time data */
          /* get CPU time in microseconds since last call */
          call TIMCPU (hooksw.after);
          temp_cpu = hooksw.after - hooksw.before;
          hook_time(ix) =  hook_time(ix) + temp_cpu;
          hooksw.before = hooksw.after;
        end;  /* end of the if exe_mon > 0 then do */
      end;  /* end of the otherwise do */
    End;  /* end of the Select */
    /* ---End of the SELECT to get the cpu timings --------  */
end;  /* end of the if (ix > 0) & (ix <=path_count) then do */
else;

/* if exit from external (main), report */
if path_type(ix) = 2  & path_block(ix) = 1  then
   Do;
     call post_hook_processing;
     free hook_time_table;
   End;
else;

/* don't invoke the debugging tool */
hook_exit_retcode = 4;
hook_exit_rsncode = 0;
/* ---------------------------------------------------------- */
Post_hook_processing: Procedure;
/* ---------------------------------------------------------- */
DECLARE Total_v_count       FLOAT DEC(06);
DECLARE Total_c_count       FLOAT DEC(06);
DECLARE Tot_vst_per_blk     FIXED BIN(31,0);
DECLARE Tot_v_pcnt_per_blk  FIXED DEC(6,4);
DECLARE Tot_cpu_per_blk     FIXED BIN(31,0);
DECLARE Tot_c_pcnt_per_blk  FIXED DEC(6,4);
DECLARE Percent_v           FIXED DEC(6,4);
DECLARE Percent_c           FIXED DEC(6,4);
DECLARE Col1                CHAR(30);
DECLARE Col2                CHAR(14);
DECLARE Col2_3              CHAR(30);
DECLARE Col3                CHAR(10);
DECLARE Col4                CHAR(14);
DECLARE Col5                CHAR(14);
DECLARE Col4_5              CHAR(28);
DECLARE Millisec_out        FIXED DEC(10,3);

 total_v_count = 0;
 total_c_count = 0;

 do jx = 1 to path_count;  /* get total hook_visits */
    total_v_count  = total_v_count + hook_visits(jx);
    total_c_count  = total_c_count + hook_time  (jx);
 end;

 do Jx = 1 to path_count;
   if path_type(jx) = 1 then
    do;  /* at block entry so print (block) headers */
      put skip list( ' ' );
      put skip list( 'Data for block ' ||
             block_name(path_block(jx)) );
      put skip list( ' ' );
      col1   = '     Statement Type';
      col2_3 = '         --- Visits ---';
      if exe_mon > 0
         then col4_5 = '--- CPU Time ---';
         else col4_5 = ' ';
      put skip list( col1 || ' ' || col2_3 || ' ' || col4_5 );
```

Figure 50 (Part 3 of 5). Sample facility 3: HOOKUP program

```
      col1 = ' ';
      col2 = '        Number';
      col3 = '  Percent';
      if exe_mon > 0 then
        do;
          col4 = '  Milliseconds';
          col5 = '  Percent';
        end;
        else
        do;
         col4 = ' ';
         col5 = ' ';
        end;
      put skip list( col1 || ' ' || col2 || ' ' || col3 ||
                                ' ' || col4 || col5);
      put skip list( ' ' );
    end;
  else;

  Select ( path_type(jx) );
    when ( 1 )
      col1 = Substr(char(path_no(jx)),4,11) || ' block entry';
    when ( 2 )
      col1 = Substr(char(path_no(jx)),4,11) || ' block exit';
    when ( 3 )
      col1 = Substr(char(path_no(jx)),4,11) || ' label';
    when ( 4 )
      col1 = Substr(char(path_no(jx)),4,11) || ' before call';
    when ( 5 )
      col1 = Substr(char(path_no(jx)),4,11) || ' after call';
    when ( 6 )
      col1 = Substr(char(path_no(jx)),4,11) || ' start of do loop';
    when ( 7 )
      col1 = Substr(char(path_no(jx)),4,11) || ' if true';
    when ( 8 )
      col1 = Substr(char(path_no(jx)),4,11) || ' if false';
    otherwise
      col1 = Substr(char(path_no(jx)),4,11);
  end;  /* end of the select */

  col2 = hook_visits(jx);
  percent_v = 100* (hook_visits(jx)/total_v_count) ;
  percent_c = 100* (hook_time  (jx)/total_c_count) ;

  /* ------ print out the cpu timings ------ */
  if exe_mon <= 0 then  /* no cpu time wanted */
     put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
  else do;  /* compute and print cpu times */
    Select;
      When (path_type(jx) = 1 & jx = 1) Do;
        /* at external block entry */
        /* initialize block counters */
        tot_vst_per_blk = hook_visits(jx);
        tot_v_pcnt_per_blk = percent_v;
        tot_cpu_per_blk    = 0;
        tot_c_pcnt_per_blk = 0;
        /* no CPU time on ext block entry line */
        put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
      End;  /* End of the When (... = 1 & jx = 1) */
```

Figure 50 (Part 4 of 5). Sample facility 3: HOOKUP program

```
        When (path_type(jx) = 2) Do;
          /* @ block exit so print cpu summary line */
          /* write the "block exit" line */
          tot_vst_per_blk = tot_vst_per_blk + hook_visits(jx);
          tot_v_pcnt_per_blk = tot_v_pcnt_per_blk + percent_v;
          tot_cpu_per_blk = tot_cpu_per_blk + hook_time  (jx);
          tot_c_pcnt_per_blk = tot_c_pcnt_per_blk + percent_c;
          if exe_mon > 1 then do;
             millisec_out = Decimal(hook_time(jx),11,3)/1000;
             put skip list ( col1 || ' ' || col2 || ' ' || percent_v
                     || ' ' || millisec_out || ' ' || percent_c);
          end;
          Else do;  /* only want cpu time on summary line */
             put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
          end;
          /* write the "Totals for Block" line */
          col1 = '     Totals for block';
          col2 = tot_vst_per_blk;
          millisec_out  = Decimal(tot_cpu_per_blk,11,3) / 1000;
          put skip list ( col1 || ' ' || col2 || ' ' ||
               tot_v_pcnt_per_blk || ' ' || millisec_out || ' ' ||
               tot_c_pcnt_per_blk);
        End;  /* End of the When (path_type(jx) = 2) */

        Otherwise do; /* cpu hook timing */
          if path_type(jx) = 1 then do;
            /* at internal block entry */
            /* initialize block counters */
            tot_vst_per_blk = hook_visits(jx);
            tot_v_pcnt_per_blk = percent_v;
            tot_cpu_per_blk    = hook_time(jx);
            tot_c_pcnt_per_blk = percent_c;
          end;
          else do;
            /* not at block entry */
            tot_vst_per_blk = tot_vst_per_blk + hook_visits(jx);
            tot_v_pcnt_per_blk = tot_v_pcnt_per_blk + percent_v;
            tot_cpu_per_blk = tot_cpu_per_blk + hook_time  (jx);
            tot_c_pcnt_per_blk = tot_c_pcnt_per_blk + percent_c;
          end;
          if exe_mon > 1 then do;
            millisec_out = Decimal(hook_time(jx),11,3)/1000;
            put skip list ( col1 || ' ' || col2 || ' ' || percent_v
                    || ' ' || millisec_out || ' ' || percent_c);
          end;  /* end of the if exe_mon > 1 then do */
          else  /* only want total cpu time for the block */
            put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
        End;  /* End of the Otherwise Do */
      End;  /* End of the Select */
    end;  /* end of the else do */
   end;  /* end of the do Jx = 1 to path_count */

  put skip list( ' ' );

  END post_hook_processing;
END Hookup;
```

Figure 50 (Part 5 of 5). Sample facility 3: HOOKUP program

```
TIMINI   CSECT
*  This program will initialize the task timer for
*  measuring the task execution (CPU) time so that calls to
*  the TIMCPU routine will return the proper result. Reg 0
*  will contain the value of 0 at exit of this program.
*  No arguments are passed to this program.
*
*   STANDARD PROLOGUE
         STM   14,12,12(13)        Save caller's registers
         LR    12,15               Get base address
         USING TIMINI,12           Establish addressability
         ST    13,SAVE+4           Forward link of save areas
         LA    10,SAVE             Our save area address
         ST    10,8(,13)           Backward link of save areas
         LR    13,10               Our save area is now current
         L     1,TIME              Get initial value
         SETT  (1)                 Set interval timer
*
*   STANDARD EPILOG
EXIT     L     13,4(,13)           Get previous save area address
         L     14,12(,13)          Restore register 14
         LM    1,12,24(13)         Restore regs 1 - 12
         SR    0,0                 Return value = 0
         MVI   12(13),X'FF'        Flag return
         SR    15,15               Return code = 0
         BR    14                  Return to caller
         DS    0D
TIME     DC    F'1000000'          Initial interval timer
*                                     value (milliseconds)
SAVE     DS    18F'0'              Our save area
         END   TIMINI
```

Figure 51. Sample facility 3: TIMINI module

```
TIMCPU   CSECT
*   This program will get the amount of time, in microseconds,
*   remaining in the interval timer that has been previously
*   set by the TIMINI routine.  The result is placed in the
*   full-word argument that is passed to this program.
*
*   STANDARD PROLOGUE
         STM   14,12,12(13)         Save caller's registers
         LR    12,15                Get base address
         USING TIMCPU,12            Establish addressability
         ST    13,SAVE+4            Forward link of save areas
         LA    10,SAVE              Our save area address
         ST    10,8(,13)            Backward link of save areas
         LR    13,10                Our save area is now current
*
         L     5,0(,1)              Get address of parameter
         TESTT ,                    Get the remaining time in R0
*
         LR    2,0                  Move it to R2
         SR    0,0                  Zero out R0
         L     1,INITTIME           Get initial timer value
         SR    1,2                  Subtract to get elapsed CPU time
         M     0,TEN                Change to microseconds
         ST    1,0(,5)              Put CPU time into parameter
*
*   STANDARD EPILOG
EXIT     L     13,4(,13)            Get previous save area address
         L     14,12(,13)           Restore register 14
         LM    1,12,24(13)          Restore registers 1 - 12
*                                     (r0 has time)
         MVI   12(13),X'FF'         Flag return
         SR    15,15                Return code = 0
         BR    14                   Return to caller
         DS    0D
CPU      DS    CL8                  Area to store remaining time
INITTIME DC    F'100000000'         Initial timer value
*                                   (hundredths of milliseconds)
TEN      DC    F'10'                Multiplier to conv to microseconds
SAVE     DS    18F'0'               Our save area
         END   TIMCPU
```

Figure 52. Sample facility 3: TIMCPU module

Chapter 10. Efficient programming

This chapter describes methods for improving the efficiency of PL/I programs.

The section titled "Efficient performance" suggests how to tune run-time performance of PL/I programs.

The "Global optimization features" section discusses the various types of global optimization performed by the compiler when OPTIMIZE(TIME) is specified. The section also contains some hints on coding PL/I programs to take advantage of global optimization.

Finally, pages 239 through 259 cover performance-related topics. Each of these sections has a title that begins "Notes about... ." They cover the following topics:

- Data elements
- Expressions and references
- Data conversion
- Program organization
- Recognition of names
- Storage control
- General statements
- Subroutines and functions
- Built-in functions and pseudovariables
- Input and output
- Record-oriented data transmission
- Stream-oriented data transmission
- Picture specification characters
- Condition handling.

Efficient performance

Because of the modularity of the PL/I libraries and the wide range of optimization performed by the compiler, many PL/I application programs have acceptable performance and do not require tuning.

This section suggests ways in which you can improve the performance of programs that do not fall into this category. Other ways to improve performance are described later in this chapter, under the notes for the various topics.

It is assumed that you have resolved system considerations (for example, the organization of the PL/I libraries), and also that you have some knowledge of the compile-time and run-time options (see Chapter 1, "Using compile-time options and facilities" and *LE/VSE Programming Guide*).

Tuning a PL/I program

Remove all debugging aids from the program.

The overhead incurred by some debugging aids is immediately obvious because these aids produce large amounts of output. However, debugging aids such as the SUBSCRIPTRANGE and STRINGRANGE condition prefixes which produce output

only when an error occurs, also significantly increase both the storage requirements and the execution time of the program.

You should also remove PUT DATA statements from the program, especially those for which no data list is specified. These statements require control blocks to describe the variables and library modules to convert the control block values to external format. Both of these operations increase the program's storage requirements.

Using the GOSTMT or GONUMBER compile-time option does not increase the execution time of the program, but will increase its storage requirements. The overhead is approximately 4 bytes per PL/I statement for GOSTMT, and approximately 6 bytes per PL/I statement for GONUMBER.

Specify run-time options in the PLIXOPT variable, rather than as parameters passed to the program initialization routines. It might prove beneficial to alter existing programs to take advantage of the PLIXOPT variable, and to recompile them. For a description of using PLIXOPT, see the *LE/VSE Programming Guide.*

After removing the debugging aids, compile and run the program with the RPTSTG run-time option. The output from the RPTSTG option gives the size that you should specify in the STACK run-time option to enable all PL/I storage to be obtained from a single allocation of system storage. For a full description of the output from the RPTSTG option, see the *LE/VSE Programming Guide.*

Manipulating source code: Many operations are handled in-line. You should determine which operations are performed in-line and which require a library call, and arrange your program to use the former wherever possible. The majority of these in-line operations are concerned with data conversion and string handling. (For conditions under which string operations are handled in-line, see Table 28 on page 243. For implicit data conversion performed in-line, see Table 29 on page 245. For conditions under which string built-in functions are handled in-line, see "In-line code" on page 229.)

In PL/I there are often several different ways of producing a given effect. One of these ways is usually more efficient than another, depending largely on the method of implementation of the language features concerned. The difference might be only one or two machine instructions, or it might be several hundred.

You can also tune your program for efficient performance by looking for alternative source code that is more efficiently implemented by the compiler. This will be beneficial in heavily used parts of the program.

It is important to realize, however, that a particular use of the language is not necessarily bad just because the correct implementation is less efficient than that for some other usage; it must be reviewed in the context of what the program is doing now and what it will be required to do in the future.

Tuning a program for a virtual storage system

The output of the compiler is well suited to the requirements of a virtual storage system. The executable code is read-only and is separate from the data, which is itself held in discrete segments. For these reasons there is usually little cause to tune the program to reduce paging. Where such action is essential, there are a number of steps that you can take. However, keep in mind that the effects of tuning are usually small.

The object of tuning for a virtual storage system is to minimize the paging; that is, to reduce the number of times the data moves from auxiliary storage into main storage and vice-versa. You can do this by making sure that items accessed together are held together, and by making as many pages as possible read-only.

When using the compiler, you can write the source program so that the compiler produces the most advantageous use of virtual storage. If further tuning is required, you can use linkage editor statements to manipulate the output of the compiler so that certain items appear on certain pages.

By designing and programming modular programs, you can often achieve further tuning of programs for a virtual storage system.

To enable the compiler to produce output that uses virtual storage effectively, take care both in writing the source code and in declaring the data. When you write source code, avoid large branches around the program. Place statements frequently executed together in the same section of the source program.

When you declare data, your most important consideration should be the handling of data aggregates that are considerably larger than the page size. You should take care that items within the aggregate that are accessed together are held together. In this situation, the choice between arrays of structures and structures of arrays can be critical.

Consider an aggregate containing 3000 members and each member consisting of a name and a number. If it is declared:

```
DCL 1 A(3000),
      2 NAME CHAR(14),
      2 NUMBER FIXED BINARY;
```

the 100th name would be held adjacent to the 100th number and so they could easily be accessed together.

However, if it is declared:

```
DCL 1 A,
      2 NAME(3000) CHAR(14),
      2 NUMBER(3000) FIXED BINARY;
```

all the names would be held contiguously followed by all the numbers, thus the 100th name and the 100th number would be widely separated.

When choosing the storage class for variables, there is little difference in access time between STATIC INTERNAL and AUTOMATIC. The storage where both types of variable are held is required during execution for reasons other than access to the variables. If the program is to be used by several independent users simultaneously, declare the procedure REENTRANT and use AUTOMATIC to provide separate copies of the variables for each user. The storage used for based

or controlled variables is not, however, required and avoiding these storage classes can reduce paging.

You can control the positioning of variables by declaring them BASED within an AREA variable. All variables held within the area will be held together.

A further refinement is possible that increases the number of read-only pages. You can declare STATIC INITIAL only those variables that remain unaltered throughout the program and declare the procedure in which they are contained REENTRANT. If you do this, the static internal CSECT produced by the compiler will be made read-only, with a consequent reduction in paging overhead.

The compiler output is a series of CSECTs (control sections). You can control the linkage editor so that the CSECTs you specify are placed together within pages, or so that a particular CSECT will be placed at the start of a page. The linkage editor statements you need to do this are given in your Linkage Editor and Loader publication.

The compiler produces at least two CSECTs for every external procedure. One CSECT contains the executable code and is known as the *program CSECT*; the other CSECT contains addressing data and static internal variables and is known as the *static CSECT*. In addition, the compiler produces a CSECT for every static external variable. A number of other CSECTs are produced for storage management. A description of compiler output is given in *LE/VSE Debugging Guide and Run-Time Messages.*

You can declare variables STATIC EXTERNAL and make procedures external, thus getting a CSECT for each external variable and a program CSECT and a static internal CSECT for each external procedure. It is possible to place a number of variables in one CSECT by declaring them BASED in an AREA that has been declared STATIC EXTERNAL.

When you have divided your program into a satisfactory arrangement of CSECTs, you can then analyze the use of the CSECTs and arrange them to minimize paging. You should realize, however, that this can be a difficult and time-consuming operation.

Global optimization features

The PL/I compiler attempts to generate object programs that run rapidly. In many cases, the compiler generates efficient code for statements in the sequence written by the programmer. In other cases, however, the compiler might alter the sequence of statements or operations to improve the performance, while producing the same result.

The compiler carries out the following types of optimization:

- Expressions:
  - Common expression elimination
  - Redundant expression elimination
  - Simplification of expressions.
- Loops:
  - Transfer of expressions from loops
  - Special-case code for DO statements.

- Arrays and structures:
  - Initialization
  - Assignments
  - Elimination of common control data.
- In-line code for:
  - Conversions
  - RECORD I/O
  - String manipulation
  - Built-in functions.
- Input/output:
  - Key handling for REGIONAL data sets
  - Matching format lists with data lists.
- Other:
  - Library subroutines
  - Use of registers
  - Analyzing run-time options during compile time (the PLIXOPT variable).

PL/I performs some of these types of optimization even when the NOOPTIMIZE option is specified. PL/I attempts full optimization, however, only when a programmer specifies the OPTIMIZE (TIME) compile-time option.

Expressions

The following sections describe optimization of expressions.

Common expression elimination

A *common expression* is an expression that occurs more than once in a program, but is intended to result in the same value each time it is evaluated. A *common expression* is also an expression that is identical to another expression, with no intervening modification to any operand in the expression. The compiler eliminates a common expression by saving the value of the first occurrence of the expression either in a temporary (compiler generated) variable, or in the program variable to which the result of the expression is assigned. For example:

```
X1 = A1 * B1;
⋮
Y1 = A1 * B1;
```

Provided that the values of A1, B1, and X1 do not change between the processing of these statements, the statements can be optimized to the equivalent of the following PL/I statements:

```
X1 = A1 * B1;
⋮
Y1 = X1;
```

Sometimes the first occurrence of a common expression involves the assignment of a value to a variable that is modified before it occurs in a later expression. In this case, the compiler assigns the value to a temporary variable. The example given above becomes:

```
Temp = A1 * B1;
X1 = Temp;
⋮
X1 = X1 + 2;
⋮
Y1 = Temp;
```

If the common expression occurs as a subexpression within a larger expression, the compiler creates a temporary variable to hold the value of the common subexpression. For example, in the expression C1 + A1 * B1, a temporary variable contains the value of A1 * B1, if this were a common subexpression.

An important application of this technique occurs in statements containing subscripted variables that use the same subscript value for each variable. For example:

```
PAYTAX(MNO)=PAYCODE(MNO)*WKPMNT(MNO);
```

The compiler computes the value of the subscript MNO only once, when the statement processes. The computation involves the conversion of a value from decimal to binary, if MNO is declared to be a decimal variable.

Redundant expression elimination

A *redundant expression* is an expression that need not be evaluated for a program to run correctly. For example, the logical expression:

```
(A=D)|(C=D)
```

contains the subexpressions (A=D) and (C=D). The second expression need not be evaluated if the first one is true. This optimization makes using a compound logical expression in a single IF statement more efficient than using an equivalent series of nested IF statements.

Simplification of expressions

There are two types of expression simplification processes explained below.

Modification of loop control variables: Where possible, the expression-simplification process modifies both the control variable and the iteration specification of a do-loop for more efficient processing when using the control variable as a subscript. The compiler calculates addresses of array elements faster by replacing multiplication operations with addition operations. For example, the loop:

```
Do I = 1 To N By 1;
  A(I) = B(I);
End;
```

assigns N element values from array B to corresponding elements in array A. Assuming that each element is 4 bytes long, the address calculations used for each iteration of the loop are:

Base(A)+(4*I) for array A, and

Base(B)+(4*I) for array B,

where Base represents the base address of the array in storage. The compiler can convert repeated multiplication of the control variable by a constant (that represents the length of an element) to faster addition operations. It converts the optimized DO statement above into object code equivalent to the following statement:

```
Do Temp = 4 By 4 To 4*N;
```

The compiler converts the element address calculations to the equivalent of:

Base(A) + Temp for array A, and

Base(B) + Temp for array B.

This optimization of a loop control variable and its iteration can occur only when the control variable (used as a subscript) increases by a constant value. Programs should not use the value of the control variable outside the loop in which it controls iteration.

Defactorization: Where possible, a constant in an array subscript expression is an offset in the address calculation. For example, PL/I calculates the address of a 4-byte element:

```
A(I+10)
```

as:

```
(Base(A)+4*10)+I*4
```

Replacement of constant expressions

Expression simplification replaces constant expressions of the form A+B or A*B, where A and B are integer constants, with the equivalent constant. For example, the compiler replaces the expression 2+5 with 7.

Replacement of constant multipliers and exponents: The expression-simplification process replaces certain constant multipliers and exponents. For example:

```
A*2 becomes A+A,
```

and

```
A**2 becomes A*A.
```

Elimination of common constants: If a constant occurs more than once in a program, the compiler stores only a single copy of that constant. For example, in the following statements:

```
Week_No = Week_No + 1;
Record_Count = Record_Count + 1;
```

the compiler stores the 1 only once, provided that Week_No and Record_Count have the same attributes.

Code for program branches

The compiler allocates base registers for branch instructions in the object program in accordance with the logical structure of the program. This minimizes the occurrence of program-addressing load instructions in the middle of deeply nested loops.

Also, the compiler arranges the branch instructions generated for IF statements as efficiently as possible. For example, a statement such as:

```
IF condition THEN GOTO label;
```

is defined by the PL/I language as being a test of *condition* followed by a branch on *false* to the statement following the THEN clause. However, when the THEN clause consists only of a GOTO statement, the statement compiles as a branch on *true* to the label specified in the THEN clause.

Loops

In addition to the optimization described in "Modification of loop control variables" on page 225, PL/I provides optimization features for loops as described in the following sections.

Transfer of expressions from loops

Where it is possible to produce an error-free run without affecting program results, optimization moves invariant expressions or statements from inside a loop to a point that immediately precedes the loop. An expression or statement occurring within a loop is said to be *invariant* if the compiler can detect that the value of the expression or the action of the statement would be identical for each iteration of the loop. A loop can be either a do-loop or a loop in a program detectable by analyzing the flow of control within the program. For example:

```
Do I = 1 To N;
  B(I) = C(I) * SQRT(N);
  P = N * J;
End;
```

This loop can be optimized to produce object code corresponding to the following statements:

```
Temp = SQRT(N);
P = N * J;
DO I = 1 TO N;
  B(I) = C(I) * Temp;
End;
```

If programmers want to use this type of optimization, they must specify REORDER for a BEGIN or PROCEDURE block that contains the loop. If a programmer does not specify the option, the compiler assumes the default option, ORDER, which inhibits optimization.

Programming considerations: The compiler transfers expressions from inside to outside a loop on the assumption that every expression in the loop runs more frequently than expressions immediately outside the loop. Occasionally this assumption fails, and the compiler moves expressions out of loops to positions in which they run more frequently than if they had remained inside the loop. For example:

```
Do I = J To K While(X(I)=0);
  X(I) = Y(I) * SQRT(N);
End;
```

The compiler might move the expression SQRT(N) out of the loop to a position in which it is *possible* for the expression to be processed more frequently than in its original position inside the loop. This undesired effect of optimization is prevented by using the ORDER option for the block in which the loop occurs.

The compiler detects loops by analyzing the flow of control. The compiler can fail to recognize a loop if programmers use label variables because of flowpaths that they know are seldom or never used. Using label variables can inadvertently inhibit optimization by making a loop undetectable.

Special case code for DO statements

Where possible for a do-loop, the compiler generates code in which the value of the control variable, and the values of the iteration specification, are contained in registers throughout loop execution. For example, the compiler attempts to maintain in registers the values of the variables I, K, and L in the following statement:

```
Do I = A To K By L;
```

This optimization uses the most efficient loop control instructions.

Arrays and structures

PL/I provides optimization for array and structure variables as described in the following sections.

Initialization of arrays and structures

When arrays and some structures that have the BASED, AUTOMATIC, or CONTROLLED storage class are to be initialized by a constant specified in the INITIAL attribute, the compiler initializes the first element of the variable by the constant. The remainder of the initialization is a single move that propagates the value through all the elements of the variable. For example, for the following declaration:

```
DCL A(20,20) Fixed Binary Init((400)0);
```

the compiler initializes array A using this method.

Structure and array assignments

The compiler implements structure and array assignment statements by single move instructions whenever possible. Otherwise, the compiler assigns values by the simplest loop possible for the operands in the declaration. For example:

```
DCL A(10),B(10), 1 S(10), 2 T, 2 U;
    A=B;
    A=T;
```

The compiler implements the first assignment statement by a single move instruction, and the second by a loop. This occurs because array T is interleaved with array U, thereby making a single move impossible.

Elimination of common control data

The compiler generates control information to describe certain program elements such as arrays. If there are two or more similar arrays, the compiler generates this descriptive information only once.

In-line code

To increase efficiency, the PL/I compiler produces *in-line* code (code that it incorporates within programs) as a substitute for calls to generalized subroutines.

In-line code for conversions

The compiler performs most conversions by in-line code, rather than by calls to the Library. The exceptions are:

- Conversions between character and arithmetic data
- Conversions from numeric character (PICTURE) data, where the picture includes characters other than 9, V, Z, or a single sign or currency character
- Conversions to numeric character (PICTURE) data, where the picture includes scale factors or floating point picture characters.

For example, the compiler converts data to PICTURE 'ZZ9V99' with in-line code.

In-line code for record I/O

For consecutive buffered files under certain conditions, in-line code implements the input and output transmission statements READ, WRITE, and LOCATE rather than calls to the Library.

In-line code for string manipulation

In-line code performs operations on many character strings (such as concatenation and assignment of adjustable, varying-length, and fixed-length strings). In-line code performs similar operations on many aligned bit strings that have a length that is a multiple of 8.

In-line code for built-in functions

The compiler uses in-line code to implement many built-in functions. INDEX and SUBSTR are examples of functions for which the compiler usually generates in-line code. TRANSLATE, VERIFY, and REPEAT are examples where the compiler generates in-line code for simple cases.

Key handling for REGIONAL data sets

In certain circumstances, avoiding unnecessary conversions between fixed binary and character-string data types simplifies key handling for REGIONAL data sets, as follows:

REGIONAL(1)

If the key is a fixed binary integer with precision (12,0) through (23,0), there is no conversion from fixed binary to character-string and back again.

REGIONAL(2) and (3)

If the key is in the form `K||I`, where `K` is a character string and `I` is fixed binary with precision (12,0) through (23,0), the rightmost eight (8) characters of the resultant string do not reconvert to fixed binary. (This conversion would otherwise be necessary to obtain the region number.)

Matching format lists with data lists

Where possible, the compiler matches format and data lists of edit-directed input/output statements at compile time. This is possible only when neither list contains repetition factors at compile time that are expressions with unknown values. This allows in-line code to convert to or from the data list item. Also, on input, PL/I can take the item directly from the buffer or, on output, place it directly into the buffer. This eliminates Library calls, except when necessary to transmit a block of data between the input or output device and the buffer. For example:

```
DCL (A,B,X,Y,Z) CHAR(25);
Get File(SYSIN) Edit(X,Y,Z) (A(25));
Put File(SYSPRINT) Edit(A,B) (A(25));
```

In this example, format list matching occurs at compile time; at run time, Library calls are required only when PL/I transmits the buffer contents to or from the input or output device.

Run-time library routines

IBM Language Environment for VSE/ESA and PL/I library routines are packaged as a set, in such a manner that a link-edited object program contains only stubs that correspond to these routines. This packaging minimizes program main storage requirements. It can also reduce the time required for loading the program into main storage.

Use of registers

The compiler achieves more efficient loop processing by maintaining in registers the values of loop variables that are subject to frequent modification. Keeping values of variables in registers, as the flow of program control allows, results in considerable efficiency. This efficiency is a result of dispensing with time-consuming load-and-store operations that reset the values of variables in their storage locations. When, after loop processing, the latest value of a variable is not required, the compiler does not assign the value to the storage location of the variable as control passes out of the loop.

Specifying REORDER for the block significantly optimizes the allocation of registers. However, because the latest value of a variable can exist in a register but not in the storage location of that variable, the values of variables reset in the block might not be the latest assigned values when a computational interrupt occurs. Specifying ORDER impedes optimizing the allocation of registers but guarantees that all values of variables are reset in the block, thereby immediately assigning values to their storage locations.

Program constructs that inhibit optimization

The following sections describe source program constructs that can inhibit optimization.

Global optimization of variables

The compiler considers 255 variables in the program for global optimization. It considers the remainder solely for local optimization.

The compiler considers explicitly-declared variables for global optimization in preference to contextually-declared variables, and then gives preference to contextually-declared variables over implicitly-declared variables. The highest preference is given to those variables declared in the final DECLARE statements in the outermost block.

If your program contains more than 255 variables, you can benefit most from the global optimization of arithmetic variables—particularly do-loop control variables and subscripting variables. You will gain little or no benefit from the optimization of string variables or program control data.

You should declare arithmetic variables in the final DECLARE statements in the outermost block rather than implicitly. You may benefit further if you eliminate declared but unreferenced variables from the program.

ORDER and REORDER options

ORDER and REORDER are optimization options specified for a procedure or begin-block in a PROCEDURE or BEGIN statement.

ORDER is the default for external procedures. The default for internal blocks is to inherit ORDER or REORDER from the containing block.

ORDER option

Specify the ORDER option for a procedure or begin-block if you must be sure that only the most recently assigned values of variables modified in the block are available for ON-units, which are entered because of computational conditions raised during the execution of statements and expressions in the block.

In a block with the ORDER option specified, the compiler might eliminate common expressions, causing fewer computational conditions to be raised during execution of the block than if common expressions had not been eliminated. But if a computational condition is raised during execution of an ORDER block, the values of variables in statements that precede the condition are the most recent values assigned when an ON-unit refers to them for the condition.

You may use other forms of optimization in an ORDER block, except for forward or backward move-out of any expression that can raise a condition. Since it would be necessary to disable all the possible conditions that might be encountered, the use of ORDER virtually suppresses any move-out of statements or expressions from loops.

REORDER option

The REORDER option allows the compiler to generate optimized code to produce the result specified by the source program, when error-free execution takes place. Move-out is allowed for any invariant statements and expressions from inside a loop to a point in the source program, either preceding or following such a loop. Thus, the statement or expression is executed once only, either before or after the loop.

More efficient execution of loops can be achieved by maintaining, in registers, the values of variables that are subject to frequent modification during the execution of the loops. When error-free execution allows, values can be kept in registers. This dispenses with time-consuming load-and-store operations needed to reset the values of variables in their storage locations. If the latest value of a variable is required after a loop has been executed, the value is assigned to the storage location of the variable when control passes out of the loop.

You can more significantly optimize register allocation if you specify REORDER for the block. However, the values of variables that are reset in the block are not guaranteed to be the latest assigned values when a computational condition is raised, since the latest value of a variable can be present in a register but not in the storage location of the variable. Thus, any ON-unit entered for a computational condition must not refer to variables set in the reorder block. However, use of the built-in functions ONSOURCE and ONCHAR is still valid in this context.

A program is in error if during execution there is a computational or system action interrupt in a REORDER block followed by the use of a variable whose value is not guaranteed.

Because of the REORDER restrictions, the only way you can correct erroneous data is by using the ONSOURCE and ONCHAR pseudovariables for a CONVERSION ON-unit. Otherwise, you must either depend on the implicit action, which terminates execution of the program, or use the ON-unit to perform error recovery and to restart execution by obtaining fresh data for computation. The second approach should ensure that all valid data is processed, and that invalid data is noted, while still taking advantage of any possible optimization. For example:

```
ON OVERFLOW PUT DATA;
DO J = 1 TO M;
DO I = 1 TO N;
X(I,J) = Y(I) + Z(J) *L + SQRT(W);
P = I*J;
END;
END;
```

When the above statements appear in a reorder block, the source code compiled is interpreted as follows:

```
ON OVERFLOW PUT DATA;
TEMP1 = SQRT(W);
DO J = 1 TO M;
TEMP2 = J;
DO I = 1 TO N;
X(I,J) = Y(I) +Z(J)*L+TEMP1;
P=TEMP2;
TEMP2=TEMP2+J;
END;
END;
```

TEMP1 and TEMP2 are temporary variables created to hold the values of expressions moved backward out of the loops, and the statement P=I*J can be simplified to P=N*M. If the OVERFLOW condition is raised, the values of the variables used in the loops cannot be guaranteed to be the most recent values assigned before the condition was raised, since the current values can be held in registers, and not in the storage location to which the ON-unit must refer.

Although this example does not show it, the subscript calculations for X, Y, and Z are also optimized.

Common expression elimination

Common expression elimination is inhibited by:

- The use in expressions of variables whose values can be reset in either an input/output or computational ON-unit.
- If a based variable is, at some point in the program, overlaid on top of a variable used in the common expression, then assigning a new value to the based variable in between the two occurrences of the common expression, inhibits optimization.

  For instance, the common expression X+Z, in the following example, is not eliminated because the based variable A which, earlier in the program, is overlaid on the variable X, is assigned a value in between the two occurrences of X+Z.

  ```
  DCL A BASED(P);
  P=ADDR(X);
      .
      .
      .
  P=ADDR(Y);
      .
      .
      .
  B=X+Z;
  P->A=2;
  C=X+Z;
  ```

- The use of aliased variables. An aliased variable is any variable whose value can be modified by references to names other than its own name. Examples are variables with the DEFINED attribute, variables used as the base for defined variables, parameters, arguments, and based variables.

Variables whose addresses are known to an external procedure by means of pointers that are either external or used as arguments are also assumed to be aliased variables.

The effect of an aliased variable does not completely prevent common expression elimination, but inhibits it slightly. For all aliased variables, the compiler builds a list of all variables that could possibly reference the aliased variable. The list is the same for each member of the list, and in a given program there can be many such lists.

When an expression containing an aliased variable is checked for its use as a common expression, the possible flow paths along which related common expressions could occur are also searched for assignments. The search is not only for the variable referenced in the expression, but also for all the members of the alias list to which that variable belongs. If the program contains an external pointer variable, it is assumed that this pointer could be set to all variables whose addresses are known to external procedures; that is, all external variables, all arguments passed to external procedures, and all variables whose addresses could be assigned to the external pointer. Thus, variables addressed by the external pointer, or by any other pointer that has a value assigned to it from the external pointer, are assumed to belong to the same alias list as the external variables.

- The form of an expression. If the expression B+C could be treated as a common expression, the compiler would not be able to detect it as a common expression in the following statement:

  ```
  D=A+B+C;
  ```

  The compiler processes the expression A+B+C from left to right. Consequently, it only recognizes the expressions A+B and (A+B)+C. However, by coding the expression D=A+(B+C), you can ensure that it is recognized.

- The scope of a common expression. In order to determine the presence of common expressions, the program is analyzed and the existence of flow units is determined. A *flow unit* is a unit of object code that can only be entered at the first instruction and left at the last instruction. A flow unit can contain several PL/I source statements; conversely, a single PL/I source statement can comprise several flow units. Common expressions are recognized across individual flow units. However, if the program flow paths between flow units are complex, the recognition of common expressions is inhibited across flow units.

Common expression elimination is assisted by these points:

- Variables in expressions should not be external, associated with external pointers, or arguments to ADDR built-in functions.
- The source program should not contain external procedures, external label variables, or label constants known to external procedures.
- Variables in expressions should not be set or accessed in ON-units, if possible.
- Expressions to be commoned or transferred must be arithmetic (for example, A+B) or string (for example, ElIF or STRING(G)) rather than compiler generated.

The type of source program construct discussed below could cause common expression elimination to be carried out when it should not be.

A PL/I block can access any element of an array or structure variable if the variable is within the block's name scope, or if it has been passed to the block. When using BASED or DEFINED variables, or pointer arithmetic under the control of the LANGLVL(SPROG) compile-time option, be careful not to access any element that has not been passed to the block, and whose containing structure or array has not been passed to the block and is not within the block's name scope. Any such attempt is invalid and may lead to unpredictable results.

In the following example, procedure X passes only the address of element A.C to procedure Y. The first assignment statement in procedure Y makes a valid change to A.C. However, other statements in procedure Y are invalid because they attempt to change the values of A.B and A.D, which procedure Y cannot legally access.

Because neither the containing structure A nor its elements B, D, and E are passed to procedure Y, elements B, D, and E are not aliased for use by procedure Y. Therefore, the compiler cannot detect that their values might change in procedure Y, so it performs common expression elimination on the expression "B + D + E" in procedure X. Changing the values of A.B and A.D in procedure Y would then cause unpredictable results.

The GOTO statement in procedure Y is another error. It causes a flow of control change between blocks that is hidden from the compiler because neither A.F nor its containing variable A are passed to procedure Y. This invalid change in the flow of control can also cause unpredictable results.

```
X: PROCEDURE OPTIONS(MAIN);
     DECLARE Y ENTRY(POINTER) EXTERNAL;
      DECLARE
       1 A,
         2 B FIXED BIN(31)  INIT(1),
         2 C FIXED BIN(31)  INIT(2),
         2 D FIXED BIN(31)  INIT(3),
         2 E FIXED BIN(31)  INIT(4),
         2 F LABEL VARIABLE INIT(L1);
     N = B + D + E;
     CALL Y(ADDR(C));
L1:  M = B + D + E;
     END X;
Y: PROCEDURE (P);
     DECLARE
       (P,Q) POINTER,
       XX    FIXED BIN(31) BASED;
     DECLARE
       1 AA   BASED,
         2 CC FIXED BIN(31),
         2 DD FIXED BIN(31),
         2 EE FIXED BIN(31),
         2 FF LABEL VARIABLE;
     P->XX = 17;              /* valid change to A.C      */
     Q     = P - 4;           /* invalid                  */
     Q->XX = 13;              /* invalid change to A.B    */
     P->DD = 11;              /* invalid change to A.D    */
     GOTO P->FF;              /* invalid flow to label L1 */
     END Y;
```

Condition handling for programs with common expression elimination

The order of most operations in each PL/I statement depends on the priority of the operators involved. However, for sub-expressions whose results form the operands of operators of lower priority, the order of evaluation is not defined beyond the rule that an operand must be fully evaluated before its value can be used in another operation. These operands include subscript expressions, locator qualifier expressions, and function references. Therefore, ON-units associated with conditions raised during the evaluation of such sub-expressions can be entered in an unpredictable order. Consequently, an expression might have several possible values, according to the order of, and action taken by, the ON-units that are entered. When a computational ON-unit is entered:

- The values of all variables set by the execution of previous statements are guaranteed to be the latest values assigned to the variables, and can be used by the ON-unit. For instance the PUT DATA statement can be used to record the values of all variables on entry to an ON-unit.
- The value of any variable set in an ON-unit resulting from a computational interrupt is guaranteed to be the latest value assigned to the variable, for any part of the program.

Where there is a possibility that variables might be modified as the result of a computational interrupt, either in the associated ON-unit, or as the result of the execution of a branch from the ON-unit, common expression elimination is inhibited. For example:

```
ON ZERODIVIDE B,C=1;
   .
   .
   .
X=A*B+B/C;
Y=A*B+D;
```

The compiler normally attempts to eliminate the re-evaluation of the sub-expression A*B in the second assignment statement. However, in this example, if the ZERODIVIDE condition is raised during the evaluation of B/C, the two values for A*B would be different. Common expression elimination is inhibited to allow for this possibility.

The above discussion applies only when the optimization option ORDER is specified or defaulted. If you do not require the guarantees described above, you can specify the optimization option REORDER. In this case, common expression elimination is not inhibited.

Transfer of invariant expressions

Transfer of invariant expressions out of loops is inhibited by:

- ORDER specified for the block. However, transfer is not entirely prevented by the ORDER option. It is only inhibited for operations that can raise computational conditions. Such operations do not include array subscript manipulation where the subscripts are calculated with logical arithmetic which cannot raise OVERFLOW.
- The use of variables whose values can be set or used by input or output statements.

- The use of variables whose values can be set in input/output or computational ON-units, or which are aliased variables.
- A complicated program flow, involving external procedures, external label variables, and label constants.
- The use of a WHILE option in a repetitive DO statement. An invariant expression can be moved out of a do-group only if it appears in a statement that would have been executed during every repetition of the do-group. The appearance of a WHILE option in the DO statement of a repetitive do-group effectively causes each statement in the do-group to be considered "not always executed," since it might prevent the do-group from being executed at all. (You could, instead, have an equivalent do-group using an UNTIL option.)
- The appearance in an expression of a variant term or sub-expression preceding an invariant term or sub-expression. For example, assume that V is variant, and that NV1 and NV2 are invariant, in a loop containing the following statement:

  ```
  X = V * NV1 * NV2;
  ```

  The appearance of V preceding the sub-expression NV1*NV2 prevents the movement of the evaluation of NV1*NV2 out of the loop. (You could, instead, write X = NV1 * NV2 * V; getting the same result while allowing the sub-expression to be moved out of the loop.)
- The appearance in an expression of a constant that is not of the same data type as subsequent invariant elements in the expression. For example, assume that NV1 and NV2 are declared FLOAT DECIMAL, and are invariant in a loop containing the following statement:

  ```
  X = 100 * NV1 * NV2;
  ```

  The constant 100 has the attributes FIXED DECIMAL. For technical reasons beyond the scope of this discussion, this mismatch of attributes prevents the movement of the entire expression. (You could, instead, write the constant as 100E0, which has the attributes FLOAT DECIMAL.)

You can assist transfer by specifying REORDER for the block.

Redundant expression elimination

Redundant expression elimination is inhibited or assisted by the same factors as for transfer of invariant expressions, described above.

Other optimization features

Optimized code can be generated for the following items:

- A do-loop control variable, except when its value can be modified either explicitly or by an ON-unit during execution of a do-loop.
- Do-loops that do not contain other do-loops. This applies only if the scope of the control variable extends beyond the block containing the do-loop, then it is given a definite value after the do-loop and before the end of the block.
- Assignment of arrays or structures, unless noncontiguous storage is used.
- Array initialization where the same value is assigned to each element, unless the array occupies noncontiguous storage.

- In-line conversions, unless they involve complicated picture or character-to-arithmetic conversions.
- In-line code for the string built-in functions SUBSTR and INDEX, unless the on-conditions STRINGSIZE or STRINGRANGE are enabled.
- Register allocation and addressing schemes, unless the program flow is complicated by use of external procedures, external label variables, or label constants known to external procedures. Optimized register usage is also inhibited by the use of aliased variables and variables that are referenced or set in an ON-unit.

Assignments and initialization

When a variable is accessed, it is assumed to have a value which was previously assigned to it, and which is consistent with the attributes of the variable. If this assumption is incorrect, the program either proceeds with incorrect data or raises the ERROR condition. Invalid values can result from failure to initialize the variable, or it can occur as a result of the variable being set in one of the following ways:

- By the use of the UNSPEC pseudovariable
- By record-oriented input
- By overlay defining a picture on a character string, with subsequent assignment to the character string and then access to the picture
- By passing as an argument an incompatible value to an external procedure, without matching the attributes of the parameter by an appropriate parameter-descriptor list
- By assignment to a based variable with different attributes, but at the same location.

Failure to initialize a variable results in the variable having an unpredictable value at run-time. Do not assume this value to be zero.

Failure to initialize a subscript can be detected by enabling SUBSCRIPTRANGE, when debugging the program (provided the uninitialized value does not lie within the range of the subscript).

Referencing a variable that has not been initialized can raise a condition. For example:

```
DCL A(10) FIXED;
A(1)=10;
PUT LIST (A);
```

The array should be initialized before the assignment statement, thus:

```
A=0;
```

Notes about data elements

- Take special care to make structures match when you intend to move data from one structure to another. This avoids conversion and also allows the structure to be moved as a unit instead of element by element.
- Use pictured data rather than character data if possible. For example, if a piece of input data contains 3 decimal digits, and neither ONSOURCE nor ONCHAR is used to correct invalid data, then:

```
DCL EXTREP CHARACTER(3),
    INTREP FIXED DECIMAL (5,0);

ON CONVERSION GOTO ERR;
INTREP = EXTREP;
```

  is less efficient than:

```
DCL EXTREP CHARACTER(3),
    PICREP PIC '999' DEF EXTREP,
    INTREP FIXED DECIMAL (5,0);

IF VERIFY(EXTREP,'0123456789')
    ¬= 0 THEN GOTO ERR;
INTREP = PICREP;
```

- Declare the FIXED BINARY (31,0) attribute for internal switches and counters, and variables used as array subscripts.
- Exercise care in specifying the precision and scale factor of variables that are used in expressions. Using variables that have different scale factors in an expression can generate additional object code that creates intermediate results.
- You should, if possible, specify and test bit strings as multiples of eight bits. However, you should specify bit strings used as logical switches according to the number of switches required. In the following examples, (a) is preferable to (b), and (b) is preferable to (c):

  Example 1:

  Single switches

```
(a)  DCL SW BIT(1) INIT('1'B);
     IF SW THEN...

(b)  DCL SW BIT(8) INIT('1'B);
     IF SW THEN...

(c)  DCL SW BIT(8) INIT ('1'B);
     IF SW = '10000000'B THEN...
```

Example 2:

Multiple switches

```
(a)  DCL B BIT(3);
     B = '111'B;
       .
       .
       .
     IF B = '111'B THEN DO;

(b)  DCL B BIT(8);
     B = '11100000'B;
       .
       .
       .
     IF B = '11100000'B THEN DO;

(c)  DCL (SW1,SW2,SW3) BIT(1);
     SW1, SW2, SW3 = '1'B;
       .
       .
       .
     IF SW1&SW2&SW3 THEN DO;
```

- If bit-string data whose length is not a multiple of 8 is to be held in structures, you should declare such structures ALIGNED.
- Varying-length strings are generally less efficient than fixed-length strings.

  Fixed-length strings are not efficient if their lengths are not known at compile time, as in the following example:

  ```
  DCL A CHAR(N);
  ```

- When storage conservation is important, use the UNALIGNED attribute to obtain denser packing of data, with a minimum of padding.
- The precision of complex expressions follows the rules for expression evaluation. For example, the precision of 1 + 1i is (2,0).
- Do not use control characters in a source program for other than their defined meanings, such as delimiting a graphic string. (Control characters are those characters whose hexadecimal value is in the range from 00 to 3F, or is FF.) Although such use is not invalid, it can make it impossible for the source program to be processed by products (both hardware and software) that implement functions for the control characters.

  For example, you might write a program to edit data for a device that recognizes hexadecimal FF embedded in the data as a control character meant to suppress display. Then, if you write:

  ```
  DCL SUPPRESS CHAR(1) STATIC INIT(' ');
  ```

  where the INIT value is hexadecimal FF, your program is not displayed on that device, because the control character is recognized and suppresses all displaying after the INIT.

  Or, suppose you are creating a record that requires 1-byte numeric fields and you choose to code the following:

  ```
  DCL X01 CHAR(1) STATIC INIT(' '); /* HEXADECIMAL'01' */
  ```

Your program can fail if it is processed by a product that recognizes control characters.

To avoid these problems, represent control character values by using X character string constants. For example:

```
DCL SUPPRESS CHAR(1) INIT('FF'X);

DCL X01 CHAR(1) INIT('01'X);
```

Thus you have the values required for execution without having control characters in the source program, you are not dependent on the bit representation of characters, and the values used are "visible" in your source program.

- In the case of unaligned fields, code is sometimes generated to move data to aligned fields. Thus, if you correctly align your data and specify the ALIGNED attribute you will avoid extraneous move instructions and improve execution speed for aligned data.

Notes about expressions and references

- Do not use operator expressions, variables, or function references where constants can be substituted.

  For example:

  ```
  DECLARE A(8);
  ```

  is more efficient than

  ```
  DECLARE A(5+3);
  ```

  and

  ```
  VERIFY(DATA,'0123456789')...;
  ```

  is more efficient than

  ```
  DCL NUMBERS CHAR(10) STATIC INIT('0123456789');
  VERIFY(DATA,NUMBERS)...;
  ```

  The preprocessor is very effective in allowing the source program to be symbolic, with expression evaluation and constant substitution taking place at compile time. The examples above could be:

  ```
  % DCL DIMA FIXED;
  % DIMA = 5+...;
  DCL A(DIMA);
  ```

  and

  ```
  % DCL NUMBERS CHAR;
  % NUMBERS='''0123456789''';
  VERIFY(DATA,NUMBERS)...;
  ```

- Use additional variables to avoid conversions. For example, consider a program in which a character variable is to be regularly incremented by 1:

  ```
  DECLARE CTLNO CHARACTER(8);
  CTLNO = CTLNO + 1;
  ```

This example requires two conversions (one of which involves a library call), while:

```
DECLARE CTLNO CHARACTER(8), DCTLNO FIXED DECIMAL;
DCTLNO = DCTLNO + 1;
CTLNO = DCTLNO;
```

requires only one conversion.

- You must ensure that the lengths of the intermediate results of string expressions do not exceed 32767 bytes for CHARACTER strings, 32767 bits for BIT strings, or 16383 double-byte characters for GRAPHIC strings. Take special care with VARYING strings, because the length used in intermediate results is the maximum possible length of the VARYING string.

  For example, the following code is incorrect because the concatenation operation in the PUT statement requires an intermediate CHARACTER string length greater than 32767 bytes. The compiler will not diagnose this error, but if the code were executed, the results would be unpredictable.

```
DECLARE
  BITSTR  BIT(32767) VARYING,
  LEN_B   FIXED BIN(31,0);

BITSTR = '00000000000000000000000000100010'B;

LEN_B  = LENGTH(BITSTR);

PUT SKIP LIST (SUBSTR(BITSTR,1,LEN_B) ||
               SUBSTR(BITSTR,1,LEN_B));
```

  For this reason, avoid declaring strings with lengths close to the limit. And if you use such strings, be especially careful not to exceed the limit when you manipulate these strings.

- Avoid mixed mode arithmetic, especially the use of character strings in arithmetic calculations.
- Concatenation operations on bit-strings are time-consuming.
- The string operations performed in-line are shown in Table 28 on page 243.
- X>Y|Z is not equivalent to X>Y|X>Z, but is equivalent to (X>Y)|Z.
- You can use parentheses to modify the rules of priority. You may also wish to use redundant parentheses to explicitly specify the order of operations.
- Array arithmetic is a convenient way of specifying an iterative computation. For example:

```
DCL A(10,20);
A=A/A(1,1);
```

  has the same effect as:

```
DCL A(10,20);
DO I=1 TO 10;
DO J=1 TO 20;
A(I,J)=A(I,J)/A(1,1);
END; END;
```

  The effect is to change the value of A(1,1) only, since the first iteration produces a value of 1 for A(1,1) if A(1,1) is not equal to zero. If you want to divide each element of A by the original value of A(1,1), you can write:

```
B=A(1,1);
A=A/B;
```

or:

```
DCL A(10,20),
    B(10,20);
B=A/A(1,1);
```

- Array multiplication does not affect matrix multiplication. For example:

```
DCL (A,B,C) (10,10);
A=B*C;
```

is equivalent to:

```
DCL (A,B,C) (10,10);
DO I=1 TO 10;
DO J=1 TO 10;
A(I,J)=B(I,J)*C(I,J);
END; END;
```

Table 28. Conditions under which string operations are handled in-line

String operation	Comments and conditions		
	Source	**Target**	**Comments**
Assign	Nonadjustable, ALIGNED, fixed-length bit string	Nonadjustable, ALIGNED, bit string	No length restriction if OPTIMIZE(TIME) is specified; otherwise maximum length of 8192 bits
	Adjustable or VARYING, ALIGNED bit string	Nonadjustable, ALIGNED bit string less than or equal to 2048 bits	Only if OPTIMIZE(TIME) is specified
	Nonadjustable, fixed-length character string	Nonadjustable character string	
	Adjustable or VARYING character string	Nonadjustable character string of length less than or equal to 256	
'and', 'not', 'or'	As for bit string assignments but no adjustable or varying-length operands are handled		
Compare	As for string assignment with the two operands taking the roles of source and target, but no adjustable or varying-length operands are handled		
Concatenate	As for string assignments, but no adjustable or varying-length source strings are handled		
STRING built-in function	When the argument is an element variable, or a nonadjustable array or structure variable in connected storage		

Note: If the target is fixed-length, the maximum length is the target length. If the target is VARYING, the maximum length is the lesser of the operand lengths.

Notes about data conversion

The data conversions performed in-line are shown in Table 29 on page 245. A conversion outside the range or condition given is performed by a library call.

Not all the picture characters available can be used in a picture involved in an in-line arithmetic conversion. The only ones allowed are:

- V and 9
- Drifting or nondrifting characters $ S and +
- Zero suppression characters Z and *
- Punctuation characters , . / and B.

For in-line conversions, pictures with this subset of characters are divided into three types:

Picture type 1:
Pictures of all 9s with an optional V and a leading or trailing sign. For example:

```
'99V999',   '99',   'S99V9',   '99V+',   '$999'
```

Picture type 2:
Pictures with zero suppression characters and optional punctuation characters and a sign character. Also, type 1 pictures with punctuation characters. For example:

```
'ZZZ',   '**/**9',   'ZZ9V.99',   '+ZZ.ZZZ',   '$///99',   '9.9'
```

Picture type 3:
Pictures with drifting strings and optional punctuation characters and a sign character. For example:

```
'$$$$',   '-,--9',   'S/SS/S9',   '+++9V.9',   '$$$9-'
```

Sometimes a picture conversion is not performed in-line even though the picture is one of the above types. This might be because:

- SIZE is enabled and could be raised.
- There is no correspondence between the digit positions in the source and target. For example, DECIMAL (6,8) or DECIMAL (5,-3) to PIC '999V99' will not be performed in-line.
- The picture can have certain characteristics that make it difficult to handle in-line. For example:
  - Punctuation between a drifting Z or a drifting * and the first 9 is not preceded by a V. For example:

    ```
    'ZZ.99'
    ```

  - Drifting or zero suppression characters to the right of the decimal point. For example:

    ```
    'ZZV.ZZ'  or  '++V++'
    ```

Table 29 (Page 1 of 2). Implicit data conversion performed in-line

Conversion		Comments and conditions
Source	**Target**	
FIXED BINARY	FIXED BINARY	None.
	FIXED DECIMAL	None.
	FLOAT	Long or short FLOAT target.
	Bit string	String must be fixed-length, ALIGNED, and with length less than or equal to 2048. STRINGSIZE condition must be disabled.
	Character string or Numeric picture	Via FIXED DECIMAL. String must be fixed-length with length less than or equal to 256 and STRINGSIZE disabled. Picture types 1, 2, or 3 when SIZE disabled.
FIXED DECIMAL	FIXED BINARY	None.
	FIXED DECIMAL	None.
	FLOAT	If q1+p1 less than or equal to 75. Long or short-FLOAT target.
	Bit string	String must be fixed-length, ALIGNED, and with length less than or equal to 2048. STRINGSIZE condition must be disabled.
	Character string	If precision = scale factor, it must be even. String must be fixed-length and length less than or equal to 256. STRINGSIZE must be disabled.
	Numeric picture	Picture types 1, 2, and 3.
FLOAT (Long or Short)	FIXED BINARY	None.
	FIXED DECIMAL	Scale factor less than 80.
	FLOAT	Source and target may be single or double length.
	Bit string	String must be fixed-length, ALIGNED, and with length less than or equal to 2048. STRINGSIZE condition must be disabled.
Bit string	FIXED BINARY	Source string must be fixed-length, ALIGNED, and with length less than or equal to 32.
	FIXED DECIMAL and FLOAT	Source must be fixed-length, ALIGNED, and with length less than 32.
	Character string	Source must be fixed-length, ALIGNED, and length 1 only.
Character	Bit string	None.
	FIXED DECIMAL	Source length 1 only. CONVERSION condition must be disabled.
	FLOAT	None.
	FIXED BINARY	None.
Character picture	Character string	String must be fixed-length with length less or equal to 256.
	Character picture	Pictures must be identical.

Table 29 (Page 2 of 2). Implicit data conversion performed in-line

Conversion		Comments and conditions
Source	**Target**	
Numeric picture type 1 and 2	FIXED BINARY	Via FIXED DECIMAL. SIZE condition disabled.
	FIXED DECIMAL	Type 2 picture without * or embedded punctuation. SIZE condition disabled.
	FLOAT	Via FIXED DECIMAL. Size condition disabled.
	Numeric picture	Picture types 1, 2, or 3. SIZE condition disabled.
Locator	Locator	None.

Notes about program organization

- Although you can write a program as a single external procedure, it is often desirable to design and write the program as a number of smaller external procedures, or modules. In PL/I, the basic units of modularity are the procedure and the begin block.

  Some of the advantages of modular programming are:

  - Program size affects the time and space required for compilation. Generally, compilation time increases more than linearly with program size, especially if the compiler has to spill text onto auxiliary storage. Also, the process of adding code to a program and then recompiling it leads to wasteful multiple-compilation of existing text. Modular programming eliminates the above possibilities.
  - If you design a procedure to perform a single function it needs to contain only the data areas for that function. Because of the nature of AUTOMATIC storage, there is less danger of corruption of data areas for other functions.
  - If you design a procedure to perform a single function, you can more easily replace it with a different version. Also, you can use the same procedure in several different applications.
  - Storage allocation for all the automatic variables in a procedure occurs when the procedure is invoked at any of its entry points. If you reduce the number of functions performed by a procedure, you can often reduce the number of variables declared in the procedure. This in turn can reduce the overall demand for storage for automatic variables.

  More important (from the efficient programming viewpoint) are the following considerations:

  - The compiler has a limitation on the number of variables that it can consider for global optimization. (The number of variables does not affect other forms of optimization.)
  - The compiler has a limitation on the number of flow units that it can consider for flow analysis and, subsequently, for global optimization.
  - If the static CSECT or the DSA exceeds 4096 bytes in size, the compiler has to generate additional code to address the more remote storage.

- If the object code for a procedure exceeds 4096 bytes in size, the compiler might have to repeatedly reset base registers.

Extra invocation of procedures increases run-time, but the use of modular programming often offsets the increase, because the additional optimization can cause significantly fewer instructions to be executed.

- Avoid *unnecessary* program segmentation and block structure. This includes all procedures, ON-units, and begin blocks that need activation and termination. The initialization and program management for these carry an overhead.

  You should assess this recommendation in conjunction with the notes on modular programming given earlier in this section.

- The procedure given initial control at run-time must have the OPTIONS(MAIN) attribute. If more than one procedure has the MAIN option, the first one encountered by the linkage editor receives control.
- Under the compiler, attempting the recursive use of a procedure that was not given the RECURSIVE attribute can result in the ERROR condition after exit from the procedure. This occurs if reference is made to automatic data from an earlier invocation of the procedure.

Notes about recognition of names

Separate external declarations for the same variable must not specify conflicting attributes (CONTROLLED EXTERNAL variables are an exception), either explicitly or by default. If this occurs, the compiler is not able to detect the conflict.

Notes about storage control

- Using self-defining data (the REFER option) enables data to be held in a compact form; it can, however, produce less-than-optimum object code. For example, with the structure:

```
DCL 1 STR BASED(P),
      2 N,
      2 BEFORE,
      2 ADJUST (NMAX REFER(N)),
        3 BELOW,
      2 AFTER;
```

  a reference to BEFORE requires one instruction to address the variable, whereas a reference to AFTER or BELOW requires approximately 18 instructions, plus a call to a resident library module.

  You can organize a self-defining structure more efficiently by ensuring that the members whose lengths are known appear before any adjustable members, and that the most frequently used adjustable members appear before those that are less frequently used. The previous example could thus be changed to:

```
DCL 1 STR BASED(P),
      2 N,
      2 BEFORE,
      2 AFTER,
      2 ADJUST (NMAX REFER(N)),
        3 BELOW;
```

- PL/I does not allow the use of the INITIAL attribute for arrays that have the attributes STATIC and LABEL, because the environmental information for the array does not exist until run-time.

  However, when compiling procedures containing STATIC LABEL arrays, you may improve performance by specifying the INITIAL attribute, provided that the number of elements in the array is less than 512. What happens under these conditions is:

  - The compiler diagnoses the invalid language (STATIC, LABEL, and INITIAL) and produces message IEL0580I, but it accepts the combination of attributes.
  - If OPT(TIME) is in effect, the compiler checks GO TO statements that refer to STATIC LABEL variables to see whether the value of the label variable is a label constant in the same block as the GO TO statement. If the value is a label constant in the same block, the compiler replaces the normal interpretative code with direct branching code. If it is not, the compiler produces message IEL0918I and the interpretative code remains unchanged.
  - If OPT(0) is in effect, or if message IEL0918I is produced, or if the array is larger than 512 elements, execution is liable to terminate with an interrupt, or go into a loop, or produce other unpredictable results.

- AUTOMATIC variables are allocated at entry to a block; sequences of the following type allocate B with the value that N had on entry to the block (not necessarily 4):

```
A: PROC;
   N=4;
   DCL B(N) FIXED;
    .
    .
    .
   END;
```

- If you specify the INITIAL attribute for an external noncontrolled variable, you must specify it, with the same value, on all the declarations for that variable. An exception to this rule is that an INITIAL attribute specified for an external variable in a compiled procedure need not be repeated elsewhere.

- String lengths, area sizes, and array bounds of controlled variables can be recomputed at the time of an ALLOCATE statement. As a result, even if the declaration asserts that a structure element (for example, X) is CHARACTER(16) or BIT(1), PL/I treats the declaration as CHARACTER(*) or BIT(*). Thus, library subroutines manipulate these strings. The compiler allocates and releases the temporary space needed for calculations (for instance, concatenation) using these strings each time the calculation is performed, since the compiler does not know the true size of the temporary space.

  An alternate way of declaring character strings of known length is to declare PICTURE '(16)X' to hold a 16-character string. Because the number “16” is not subject to change at ALLOCATE time, better code results.

Notes about statements

- If a GOTO statement references a label variable, it is more efficient if the label constants that are the values of the label variable appear in the same block as the GOTO statement.
- When storage conservation is important, avoid the use of iterative do-groups with multiple specifications. The following is inefficient in terms of storage requirements:

```
DO I = 1,3,-5,15,31;
  .
  .
  .
END;
```

- A repetitive do-group is not executed if the terminating condition is satisfied at initialization:

```
I=6;
DO J=I TO 4;
   X=X+J;
END;
```

  This group does not alter X, since BY 1 is implied. Iterations can step backwards, and if BY -1 was specified, three iterations would take place.

- Expressions in a DO statement are assigned to temporaries with the same characteristics as the expression, not the variable. For example:

```
DCL A DECIMAL FIXED(5,0);
    A=10;
    DO I=1 TO A/2;
     .
     .
     .
    END;
```

  This loop is not executed, because A/2 has decimal precision (15,10), which, on conversion to binary (for comparison with I), becomes binary (31,34).

  Five iterations result if the DO statement is replaced by:

```
ITEMP=A/2;
DO I=1 TO ITEMP;
```

  or:

```
DO I=1 TO PREC(A/2,6,1)
```

- Upper and lower bounds of iterative do-groups are computed only once, even if the variables involved are reassigned within the group. This applies also to the BY expression.

  Any new values assigned to the variables involved take effect only if the do-group is started again.

- In a do-group with both a control variable and a WHILE option, the evaluation and testing of the WHILE expression is carried out only after determination (from the value of the control variable) that iteration can be performed. For example, the following group is executed at most once:

```
DO I=1 WHILE(X>Y);
 .
 .
 .
END;
```

- If the control variable is used as a subscript within the do-group, ensure that the variable does not run beyond the bounds of the array dimension. For instance:

```
DECLARE A(10);
DO I = 1 TO N;
 .
 .
 .
A(I) = X;
 .
 .
 .
END;
```

  If N is greater than 10, the assignment statement can overwrite data beyond the storage allocated to the array A. A bounds error can be difficult to find, particularly if the overwritten storage happens to contain object code. You can detect the error by enabling SUBSCRIPTRANGE.

  You should explicitly declare the control variable as FIXED BIN (31,0), because the default is FIXED BIN (15,0) which cannot hold fullword subscripts.

- If a Type 2 DO specification includes both the WHEN and UNTIL options, the DO statement provides for repetitive execution, as defined by the following:

```
LABEL:  DO WHILE (expression1)
           UNTIL (expression2)
        statement-1
        .
        .
        .
        statement-n
        END;
NEXT:   statement /* STATEMENT FOLLOWING THE DO GROUP */
```

  The above is exactly equivalent to the following expansion:

```
LABEL:  IF (expression 1) THEN; ELSE
          GO TO NEXT;
        statement-1
        .
        .
        .
        statement-n
LABEL2: IF (expression2) THEN; ELSE
          GO TO LABEL;
NEXT:   statement /* STATEMENT FOLLOWING THE DO GROUP */
```

  The statement GO TO LABEL; replaces the IF statement at label LABEL2 in the expansion if the UNTIL option is omitted.

  A null statement replaces the IF statement at label LABEL. Note that, if the WHILE option is omitted, statements 1 through n are executed at least once.

- If the Type 3 DO specification includes the TO and BY options, the action of the do-group is defined by the following:

```
LABEL:  DO variable=
            expression1
            TO expression2
            BY expression3
            WHILE (expression4)
            UNTIL(expression5);
            statement-1
            .
            .
            .
            statement-m
LABEL1:  END;
NEXT:    statement
```

For a variable that is not a pseudovariable, the above action of the do-group definition is exactly equivalent to the following expansion, in which 'p' is a compiler-created pointer, 'v' is a compiler-created based variable based on 'p' and with the same attributes as 'variable,' and 'e*n*' (*n*=1, 2, or 3) is a compiler-created variable: For example:

```
LABEL:  p=ADDR(variable);
        e1=expression1;
        e2=expression2;
        e3=expression3;
        v=e1;
LABEL2: IF (e3>=0)&(v>e2)|
           (e3<0)&(v<e2)
           THEN GO TO NEXT;
        IF (expression4) THEN;
           ELSE GO TO NEXT;
        statement-1
        .
        .
        .
        statement-m
LABEL1: IF (expression5) THEN
        GO TO NEXT;
LABEL3: v=v+e3;
        GO TO LABEL2;
NEXT:   statement
```

If the specification includes the REPEAT option, the action of the do-group is defined by the following:

```
LABEL:  DO variable=
            expression1
            REPEAT expression6
            WHILE (expression4)
            UNTIL(expression5);
            statement-1
            .
            .
            .
            statement-m
LABEL1:  END;
NEXT:    statement
```

For a variable that is not a pseudovariable, the above action of the do-group definition is exactly equivalent to the following expansion:

```
LABEL:  p=ADDR(variable);
        e1=expression1;
        v=e1;
LABEL2: ;
        IF (expression4) THEN;
          ELSE GO TO NEXT;
          statement-1
          .
          .
          .
          statement-m
LABEL1: IF (expression5) THEN
          GO TO NEXT;
LABEL3: v=expression6;
          GO TO LABEL2;
NEXT:   statement
```

Additional rules for the above expansions follow:

- The above expansion only shows the result of one specification. If the DO statement contains more than one specification, the statement labeled NEXT is the first statement in the expansion for the next specification. The second expansion is analogous to the first expansion in every respect. Note, however, that statements 1 through m are not actually duplicated in the program.
- Omitting the WHILE clause also omits the IF statement immediately preceding statement-1 in each of the expansions.
- Omitting the UNTIL clause also omits the IF statement immediately following statement-m in each of the expansions.

Notes about subroutines and functions

- You should consider the precision of decimal constants when such constants are passed. For example:

```
CALL ALPHA(6);

ALPHA:PROCEDURE(X);
      DCL X FIXED DECIMAL;
      END;
```

If ALPHA is an external procedure, the above example is incorrect because X is given default precision (5,0), while the decimal constant, 6, is passed with precision (1,0).

- When a procedure contains more than one entry point, with different parameter lists on each entry, do not make references to parameters other than those associated with the point at which control entered the procedure. For example:

```
A:PROCEDURE(P,Q);
   P=Q+8; RETURN;
B: ENTRY(R,S);
   R=P+S;       /* THE REFERENCE TO P IS AN ERROR */
   END;
```

When an EXTERNAL ENTRY is declared with no parameter descriptor list or with no parameters, no matching between parameters and arguments will be done. Therefore, if any arguments are specified in a CALL to such an entry point, no diagnostic message will be issued.

If one of the following examples is used:

```
DECLARE X ENTRY EXTERNAL;
```

or

```
DECLARE X ENTRY() EXTERNAL;
```

any corresponding CALL statement, for example:

```
CALL X(parameter);
```

will not be diagnosed, although it might be an error.

Notes about built-in functions and pseudovariables

For efficiency, do not refer to the DATE built-in function more than once in a run. For example, instead of:

```
PAGEA= TITLEA||DATE;
PAGEB= TITLEB||DATE;
```

it is more efficient to write:

```
DTE=DATE;
PAGEA= TITLEA||DTE;
PAGEB= TITLEB||DTE;
```

Table 30 on page 254 shows the conditions under which string built-in functions are performed in-line.

Assignments made to a varying string by means of the SUBSTR pseudovariable do not set the length of the string. A varying string initially has an undefined length, so that if all assignments to the string are made using the SUBSTR pseudovariable, the string still has an undefined length and cannot be successfully assigned to another variable or written out.

When UNSPEC is used as a pseudovariable, the expression on the right is converted to a bit string. Consequently, the expression must not be invalid for such conversion. For example, if the expression is a character string containing characters other than 0 or 1, a conversion error will result.

If both operands of the MULTIPLY built-in function are FIXED DECIMAL and have precision greater than or equal to 14, unpredictable results can occur if SIZE is enabled.

If an array or structure, whose first element is an unaligned bit i1.arrays, first element of, unaligned bit string restriction string that does not start on a byte boundary, is used as an argument in a call to a subroutine or a built-in function, the results are unpredictable.

Table 30. Conditions under which string built-in functions are handled in-line

String function	Comments and conditions
BIT	Always
BOOL	The third argument must be a constant. The first two arguments must satisfy the conditions for 'and', 'or', and 'not' operations in Table 28 on page 243.
CHAR	Always
HIGH	Always
INDEX	Second argument must be a nonadjustable character string less than 256 characters long.
LENGTH	Always
LOW	Always
REPEAT	Second argument must be constant.
SUBSTR	STRINGRANGE must be disabled.
TRANSLATE	First argument must be fixed-length; second and third arguments must be constant.
UNSPEC	Always
VERIFY	First argument must be fixed-length: • If CHARACTER it must be less than or equal to 256 characters, • If BIT it must be ALIGNED, less than or equal to 2048 bits. Second argument must be constant.

Notes about input and output

- Allocate sufficient buffers to prevent the program from becoming I/O bound, and use blocked output records. However, consider the impact on other programs executing within the system.
- When a number of data sets are accessed by a single statement, use of a file variable rather than the TITLE option can improve program efficiency by allowing a file for each data set to remain open for as long as the data set is required by the program. Using the TITLE option requires closing and reopening a file whenever the statement accesses a new data set.
- The OPEN statement is executed by library routines that are loaded dynamically at the time the OPEN statement is executed. Consequently,

run-time could be improved if you specify more than one file in the same OPEN statement, since the routines need be loaded only once, regardless of the number of files being opened. However, opening multiple files can temporarily require more internal storage than might otherwise be needed.

As with the OPEN statement, closing more than one file with a single CLOSE statement can save run-time, but it can require the temporary use of more internal storage than would otherwise be needed.

Notes about record-oriented data transmission

- When you create or access a CONSECUTIVE data set, use file and record variable declarations that generate in-line code, if possible. Details of the declarations are given in Chapter 6, "Defining and using consecutive data sets" on page 86.
- Conversion of source keys for REGIONAL data sets can be avoided if the following special cases are observed:
  - For REGIONAL(1): When the source key is a fixed binary element variable or constant, use precision in the range (12,0) to (23,0).
  - For REGIONAL(2) and (3): When the source key is of the form (character-expression||r), where r is a fixed binary element variable or constant, use precision in the range (12,0) to (23,0).
- A pointer set in READ SET or LOCATE SET is not valid beyond the next operation on the file, or beyond a CLOSE statement. In OUTPUT files, you can mix WRITE and LOCATE statements freely.
- When you need to rewrite a record that was read into a buffer (using a READ SET statement that specifies a SEQUENTIAL UPDATE file) and then updated, you can use the REWRITE statement without a FROM option. A REWRITE after a READ SET always rewrites the contents of the last buffer onto the data set.

  For example:

```
3  READ FILE (F) SET (P);
             .
             .
             .
5  P->R = S;
             .
             .
             .
7  REWRITE FILE (F);
             .
             .
             .
11  READ FILE (F) INTO (X);
             .
             .
             .
15  REWRITE FILE (F);
             .
             .
             .
19  REWRITE FILE (F) FROM (X);
```

Statement 7 rewrites a record updated in the buffer.

Statement 15 does not change the record on the data set at all.

Statement 19 raises the ERROR condition, since there is no preceding READ statement.

- There is one case where it is not possible to check for the KEY condition on a LOCATE statement until transmission of a record is attempted. This is when there is insufficient room in the specified region to output the record on a REGIONAL(3) V- or U-format file. Neither the record raising the condition nor the current record is transmitted.

  If a LOCATE is the last I/O statement executed before the file closes, the record is not transmitted and the condition can be raised by the CLOSE statement.

Notes about stream-oriented data transmission

- A common programming practice is to put the format list used by edit-directed input/output statements into a FORMAT statement. The FORMAT statement is then referenced from the input/output statement by means of the R-format item. For example:

```
DECLARE NAME CHARACTER(20), MANNO DEC FIXED(5,0);
   .
   .
   .
PUT EDIT (MANNO,NAME)(R(OUTF));
   .
   .
   .
OUTF:FORMAT(F(8),X(5),A(20));
```

  Programming in this way reduces the level of optimization that the compiler is able to provide for edit-directed input/output. If the format list repeats in each input/output statement:

```
PUT EDIT (MANNO,NAME) (F(8),X(5),A(20));
```

  the compiler is able to generate more efficient object code which calls fewer library modules, with a consequent saving in run-time and load module size.

- In STREAM input/output, do as much as possible in each input/output statement. For example:

```
PUT EDIT ((A(I) DO I = 1 TO 10))(...;
```

  is more efficient than:

```
DO I = 1 TO 10;
   PUT EDIT (A(I))(...
END;
```

- Use edit-directed input/output in preference to list- or data-directed input/output.

- Consider the use of overlay defining to simplify transmission to or from a character string structure. For example:

```
DCL 1 IN,
      2 TYPE CHAR(2),
      2 REC,
        3 A CHAR(5),
        3 B CHAR(7),
        3 C CHAR(66);
 GET EDIT(IN) (A(2),A(5),A(7),A(66));
```

  In the above example, each format-item/data-list-item pair is matched separately, code being generated for each matching operation. It would be more efficient to define a character string on the structure and apply the GET statement to the string:

```
DCL STRNG CHAR(80) DEF IN;
GET EDIT (STRNG) (A(80));
```

- When an edit-directed data list is exhausted, no further format items will be processed, even if the next format item does not require a matching data item. For example:

```
DCL A FIXED(5),
    B FIXED(5,2);
GET EDIT (A,B) (F(5),F(5,2),X(70));
```

  The X(70) format item is not processed. To read a following card with data in the first ten columns only, the SKIP option can be used:

```
GET SKIP EDIT (A,B) (F(5), F(5,2));
```

- The number of data items represented by an array or structure name appearing in a data list is equal to the number of elements in the array or structure; thus if more than one format item appears in the format list, successive elements will be matched with successive format items.

  For example:

```
DCL 1 A,
      2 B CHAR(5),
      2 C FIXED(5,2);
   .
   .
   .
PUT EDIT (A) (A(5),F(5,2));
```

  B is matched with the A(5) item, and C is matched with the F(5,2) item.

- If the ON statement:

```
ON ERROR PUT DATA;
```

  is used in an outer block, you must remember that variables in inner blocks are not known and, therefore, are not dumped. You might wish, therefore, to repeat the ON-unit in all inner blocks during debugging.

- When you use PUT DATA without a data-list, every variable known at that point in the program is transmitted in data-directed output format to the specified file. Users of this facility, however, should note that:
  - Uninitialized FIXED DECIMAL data might raise the CONVERSION condition or the ERROR condition.

- Unallocated CONTROLLED data causes arbitrary values to print and, in the case of FIXED DECIMAL, can raise the CONVERSION condition or the ERROR condition.

- An output file generated with PUT LIST contains a blank after the last value in each record. If the file is edited with an edit program that deletes trailing blanks, then values are no longer separated and the file cannot be processed by GET LIST statements.

Notes about picture specification characters

- In a PICTURE declaration, the V character indicates the scale factor, but does not in itself produce a decimal point on output. The point picture character produces a point on output, but is purely an editing character and does not indicate the scale factor. In a decimal constant, however, the point does indicate the scale factor. For example:

```
DCL A PIC'99.9',
    B PIC'99V9',
    C PIC'99.V9';
A,C=45.6;
B=7.8;
PUT LIST (A,B,C);
```

  This puts out the following values for A, B, and C, respectively:

```
04.5    078     45.6
```

  If these values were now read back into the variables by a GET LIST statement, A, B, and C would be set to the following respective values:

```
004     780    45.6
```

  If the PUT statement were then repeated, the result would be:

```
00.4    780     45.6
```

- Checking of a picture is performed only on assignment into the picture variable:

```
DCL A PIC'999999',
    B CHAR(6) DEF A,
    C CHAR(6);
B='ABCDEF';
C=A;       /* WILL NOT RAISE CONV CONDITION */
A=C;       /* WILL RAISE CONV */
```

  Note also (A, B, C as declared above):

```
A=123456;  /* A HAS VALUE 123456   */
           /* B HAS VALUE '123456' */
C=123456;  /* C HAS VALUE 'bbb123' */
C=A;       /* C HAS VALUE '123456' */
```

Notes about condition handling

- Even though a condition was explicitly disabled by means of a condition prefix, a lot of processing can be required if the condition is raised. For example, consider a random number generator in which the previous random number is multiplied by some factor and the effects of overflow are ignored:

  ```
  DECLARE (RANDOM,FACTOR) FIXED BINARY(31,0);
  (NOFOFL):RANDOM=RANDOM*FACTOR;
  ```

  If the product of RANDOM and FACTOR cannot be held as FIXED BINARY(31,0), a hardware program-check interrupt occurs. This has to be processed by both the operating system interrupt handler and the PL/I error handler before it can be determined that the FIXEDOVERFLOW condition was disabled.

- If possible, avoid using ON-units for the FINISH condition. These increase the time taken to terminate a PL/I program, and can also cause loops, abends, or other unpredictable results.

- After debugging, disable any normally-disabled conditions that were enabled for debugging purposes by removing the relevant prefixes, rather than by including NO-condition prefixes. For instance, disable the SIZE condition by removing the SIZE prefix, rather than by adding a NOSIZE prefix. The former method allows the compiler to eliminate code that checks for the condition, whereas the latter method necessitates the generation of extra code to prevent the checks being carried out.

- If the specified action is to apply when the named condition is raised by a given statement, the ON statement must be executed before that statement. The statements:

  ```
  GET FILE (ACCTS) LIST (A,B,C);
  ON TRANSMIT (ACCTS) GO TO TRERR;
  ```

  result in the ERROR condition being raised in the event of a transmission error during the first GET operation, and the required branch is not taken (assuming that no previous ON statement applies). Furthermore, the ON statement is executed after each execution of the GET statement.

- At normal exit from an AREA ON-unit, the implicit action is to try again to make the allocation. As a result, the ON-unit is entered again, and an indefinite loop is created. To avoid this, you should modify the amount allocated in the ON-unit, for example, by using the EMPTY built-in function or by changing a pointer variable.

- Do not use ON-units to implement the program's logic; use them only to recover from truly exceptional conditions. Whenever an ON-unit is entered, considerable error-handling overhead is incurred. To implement the logic, you should perform the necessary tests, rather than use the compiler's condition-detecting facilities.

  For example, in a program using record-oriented output to a keyed data set, you might wish to eliminate certain keys because they would not fit into the limits of the data set. You can rely on the raising of the KEY condition to detect unsuitable keys, but it is considerably more efficient to test each key yourself.

Part 5. Using interfaces to other products

Chapter 11. Using the Sort program

The PL/I compiler provides an interface called PLISRT*x* (*x* = A, B, C, or D) that allows you to make use of a sort utility program to provide sorting functions within your program. The sorting functions themselves are not provided by PL/I; your system needs to have a suitable sort/merge product available. The PLISRT*x* routines provide the interface between your program and the Sort program.

In this chapter, the terms "the Sort program" and "Sort" refer to the sort/merge utility that your installation uses. This could be one of the IBM-supplied programs, DOS/VS Sort/Merge II or DFSORT/VSE, or a product from another supplier that uses a compatible interface.

To use the Sort program with PLISRT*x*, you must:

1. Include a call to one of the entry points of PLISRT*x*, passing it the information on the fields to be sorted. This information includes the length of the records, the amount of storage used, 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 your JCL.

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 other processing to be done before or after the sort itself. This allows you to develop a complete data extraction, sorting, and printing operation, which will be handled entirely within the one program. 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 either of the IBM Sort programs, DOS/VS Sort/Merge II or DFSORT/VSE, or a program with the same interface. The following material applies to the IBM Sort programs. Because you can use programs other than these, the actual capabilities and restrictions vary. For these capabilities and restrictions, see the relevant 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 require.

To determine the PLISRT*x* 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, but using PL/I subroutines gives you more flexibility and more function,

and enables 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 272.

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 PLISRT*x*. 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-length records

Specify these on the RECORD statement, which is the second argument to PLISRT*x*.

Finally, you must decide on the amount of main and auxiliary storage you must allow for the Sort program. For further details, see the reference material for your sort product.

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 PLISRT*x*. 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 PLISRT*x* statement must contain an argument list giving the Sort program information about the data set to be sorted, the fields on which it is to be sorted, the name of a variable into which Sort 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 PLISRT*x* argument list and the choice of PLISRT*x* 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 53 is a simplified flowchart showing this operation.

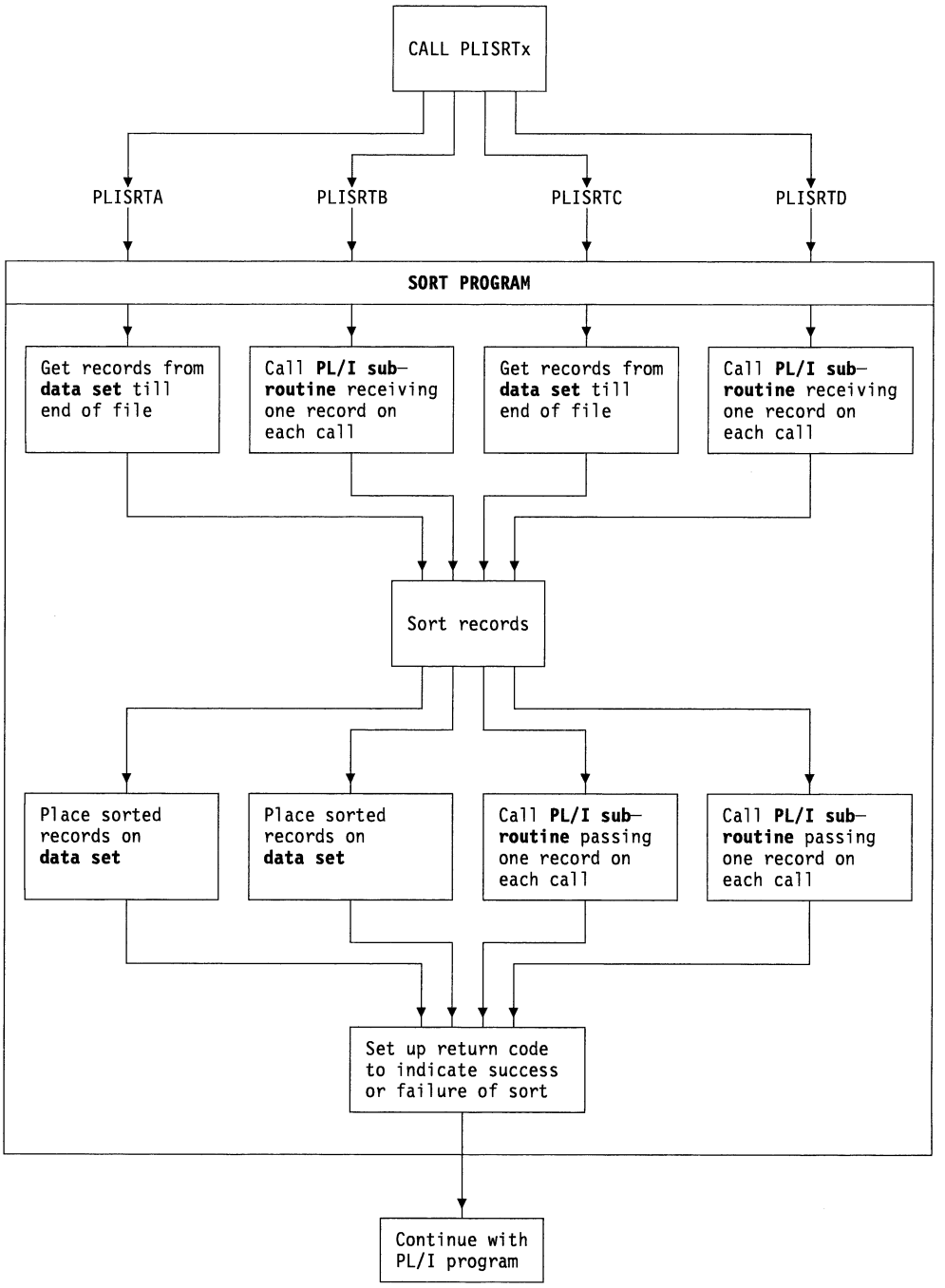


Figure 53. Flow of control for sort program

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 fields

The SORT statement is the first argument to PLISRT*x*. 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
define the sorting fields. 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.

The maximum total length of the sorting fields is restricted. The current allowed length is 3092 bytes, and all sorting fields must be within 4092 bytes of the start of the record.

start is the starting position within the record, in bytes. For fixed-length records the first byte of the record is 1, and for varying-length records you must include the 4-byte length prefix, making 5 the first byte of data.

length is the length in bytes of the sorting field. The length of sorting fields is restricted according to the format of the data in the field.

form is the format of the data. This is the format assumed for the purpose of sorting. The main data types and the restrictions on their lengths are shown below. Additional data types are available for special-purpose sorts. See the relevant publication for your sort product for details.

Code	Data type and length
CH	character 1–256
ZD	zoned decimal signed 1–256
PD	packed decimal signed 1–32
FI	fixed point, signed 1–256
BI	binary, unsigned 1–256
FL	floating-point, signed 1–256

The sum of the lengths of all fields must not exceed 3072 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)

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. Some of the other options are:

CKPT

specifies that checkpoints are to be taken. If you use this option, you must provide a SORTCKP data set on SYS000 for the Sort program to use.

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 NOEQUALS. The default option is chosen when Sort is installed. The IBM-supplied default is NOEQUALS.

Specifying the records to be sorted

Use the RECORD statement as the second argument to PLISRT*x*. The syntax of the RECORD statement must be a character string expression which, when evaluated, takes the syntax shown below:

```
'bRECORDbTYPE=rectype[,LENGTH=(length1[,,,length4,length5])]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. 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.

length1 is the length of the record to be sorted. For VSAM data sets sorted as varying records it is the maximum record size+4.

length4 specifies the minimum length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

length5 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 the records that the program can handle depends on the amount of main storage available. The length of a record can never exceed the maximum length specified by the user. The maximum record length with variable length records is 32767 bytes, and for fixed length records it is 32761 bytes.

For spanned records, maximum lengths are similar. Conditions such as control fields of different formats, large numbers of control fields, or large numbers of work data sets reduce the length of the records that can be sorted using a given amount of storage.

The actual maximum depends on storage availability and the track length of the device. For information about calculating storage requirements, see the relevant publication for your sort product.

Specifying the sort options

The OPTION statement is the third argument to PLISRT*x*. This must be a character string expression which, when evaluated, takes the syntax shown below:

```
'bOPTIONb[parameters]b'
```

Some of the parameters of the OPTION statement deserve special mention:

PRINT
PRINT=NONE should be specified as the first parameter. This is because the sort program will print messages on SYSLST, and this will conflict with other uses of the file. If it is necessary to use PRINT=ALL or PRINT=CRITICAL, then SYSLST must be closed when PLISRT*x* is called, and the sort user exits must not use SYSLST. If SYSLST is being used for any output from the Sort program, and for some reason the PL/I error routines get control, an ABEND will occur.

ROUTE
You can specify ROUTE=LOG or ROUTE=xxx to have messages sent to destinations other than SYSLST. If you do this, you can specify PRINT=ALL without conflicting with other SYSLST output, because the sort messages will go to the file specified in the ROUTE option instead of to SYSLST.

DUMP
You should not specify DUMP because, if a dump is produced, it will *always* go to SYSLST.

STORAGE
You should always specify STORAGE unless the minimum size of storage is acceptable for the sort/merge program. The minimum requirement is 32K bytes, but performance can generally be improved if more storage is available. The documentation for your sort product will contain information to help you estimate storage requirements. The format of the STORAGE parameter is STORAGE=n or STORAGE=nK.

Describing the sort input

The INPFIL statement is an optional argument to PLISRT*x*. This is a character string expression which, when evaluated, takes the syntax shown below:

```
'bINPFILb[parameters]b'
```

For PLISRTA and PLISRTC, where the sort reads its input from a data set, the INPFIL statement defines the characteristics of the data set. For example:

```
' INPFIL BLKSIZE=800 '
```

For PLISRTB and PLISRTD, where the sort input is supplied by a PL/I procedure, the INPFIL statement should only specify the EXIT parameter, as:

```
' INPFIL EXIT '
```

Describing the sort output

The OUTFIL statement is an optional argument to PLISRT*x*. This is a character string expression which, when evaluated, takes the syntax shown below:

```
'bOUTFILb[parameters]b'
```

For PLISRTA and PLISRTB, where the sort program writes its output directly to a data set, the OUTFIL statement defines the characteristics of the data set. For example:

```
' OUTFIL BLKSIZE=100 '
```

For PLISRTC and PLISRTD, where the sort passes each sorted record to a PL/I procedure, the OUTFIL statement should only specify the EXIT parameter, as:

```
' OUTFIL EXIT '
```

Calling the Sort program

When you have determined the points described above, you are in a position to write the CALL PLISRT*x* statement. Table 31 on page 269 describes the four different entry points and the arguments to be used with each entry point, and Table 32 on page 269 describes the format and content of each of the arguments.

Table 31. 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, OPTION statement, return code [,INPFIL statement[,OUTFIL statement]])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(SORT statement, RECORD statement, OPTION statement, return code, input routine [,INPFIL statement[,OUTFIL statement]])
PLISRTC Sort input: data set Sort output: PL/I subroutine	(SORT statement, RECORD statement, OPTION statement, return code, output routine [,INPFIL statement[,OUTFIL statement]])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(SORT statement, RECORD statement, OPTION statement, return code, input routine, output routine [,INPFIL statement[,OUTFIL statement]])

Table 32. The arguments to PLISRTx (x = A, B, C, or D)

Argument	Description
SORT statement	Character string expression containing the Sort program SORT statement. Describes sorting fields and format. See "Specifying the sorting fields" on page 265.
RECORD statement	Character string expression containing the Sort program RECORD statement. Describes the length and format of records to be sorted. See "Specifying the records to be sorted" on page 266.
OPTION statement	Character string expression containing the Sort program OPTION statement. Describes the options to be used by the Sort program. See "Specifying the sort options" on page 267.
Return code	Fixed binary variable of precision (31,0) in which Sort places a return code when it has completed. The meaning of the return code is: 0=Sort successful 16=Sort failed
Input routine	(PLISRTB and PLISRTD only) Name of the PL/I external or internal procedure used to supply the records for the Sort program at sort exit 15.
Output routine	(PLISRTC and PLISRTD only) Name of the PL/I external or internal procedure to which Sort passes the sorted records from sort exit 35.
INPFIL statement	Character string expression containing the Sort program INPFIL statement. Describes the format of the input file. See "Describing the sort input" on page 268.
OUTFIL statement	Character string expression containing the Sort program OUTFIL statement. Describes the format of the output file. See "Describing the sort output" on page 268.

The examples below indicate the form that the CALL PLISRT*x* statement normally takes.

Example 1

A call to PLISRTA sorting 80-byte records from SORTIN1 to SORTOUT using 128K bytes 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) ',
              ' OPTION PRINT=NONE,STORAGE=128K ',
              RETCODE,
              ' INPFIL BLKSIZE=800 ',
              ' OUTFIL BLKSIZE=800 ');
```

Example 2

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) ',
              ' OPTION PRINT=NONE,STORAGE=128K ',
              RETCODE,
              ' INPFIL BLKSIZE=800 ',
              ' OUTFIL BLKSIZE=800 ');
```

Example 3

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) ',
              ' OPTION PRINT=NONE,STORAGE=128K ',
              RETCODE,
              PUTIN,
              ' INPFIL EXIT ',
              ' OUTFIL BLKSIZE=800 ');
```

Example 4

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.

```
CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(84) ',
              ' OPTION PRINT=NONE,STORAGE=128K ',
              RETCODE,
              PUTIN,          /*input routine (sort exit 15) */
              PUTOUT,         /*output routine (sort exit 35)*/
              ' INPFIL EXIT ',
              ' OUTFIL EXIT ');
```

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 PLISRT*x*. It then returns control to the statement that follows the CALL PLISRT*x* statement. The value returned indicates the success or failure of the sort as follows:

0	Sort successful
16	Sort failed

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 PLISRT*x* 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

You must specify the data sets required by Sort through the JCL for your job step. This is in addition to the JCL for the PL/I program itself, and its own data sets. Table 33 shows the file names and symbolic device names used by the sort program, and the conditions under which these data sets are needed.

Table 33. File names and symbolic devices used by the Sort program

Use of device	File name	Symbolic device name when:			
		Sort reads input and writes output	**User routine at E15 reads input**	**User routine at E35 writes output**	**User routines read input and write output**
Output	SORTOUT	SYS001	SYS001		
Input[1]	SORTIN1	SYS002		SYS001	
	.	.		.	
	.	.		.	
	SORTIN*n*	SYS(*n*+1)		SYS(*n*)	
Work[2]	SORTWK1	SYS(*n*+2)	SYS002	SYS(*n*+1)	SYS001
	.	.	.	.	.
	.	.	.	.	.
	SORTWK*m*	SYS(*n*+*m*+1)	SYS(*m*+1)	SYS(*n*+*m*)	SYS(*m*)
Checkpoint	SORTCKP	SYS000	SYS000	SYS000	SYS000

Notes:

1. *n*=the number of input files, as specified in the FILES parameter of the SORT statement.
2. *m*=the number of work files, as specified in the WORK parameter of the SORT statement.

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 57 on page 276. Other interfaces require either the input handling routine or the output handling routine, or both.

Data input and output handling routines

The input and output handling routines are called by Sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRT*x*, they behave in the same way as ordinary internal procedures in respect of scope of names. The input and output procedure names must themselves be known in the procedure that makes the call to PLISRT*x*.

The routines are called individually for each record required by Sort or passed from Sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures 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 Figure 58 on page 277 and Figure 60 on page 279, 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 PLISRT*x*. 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 54 on page 273.

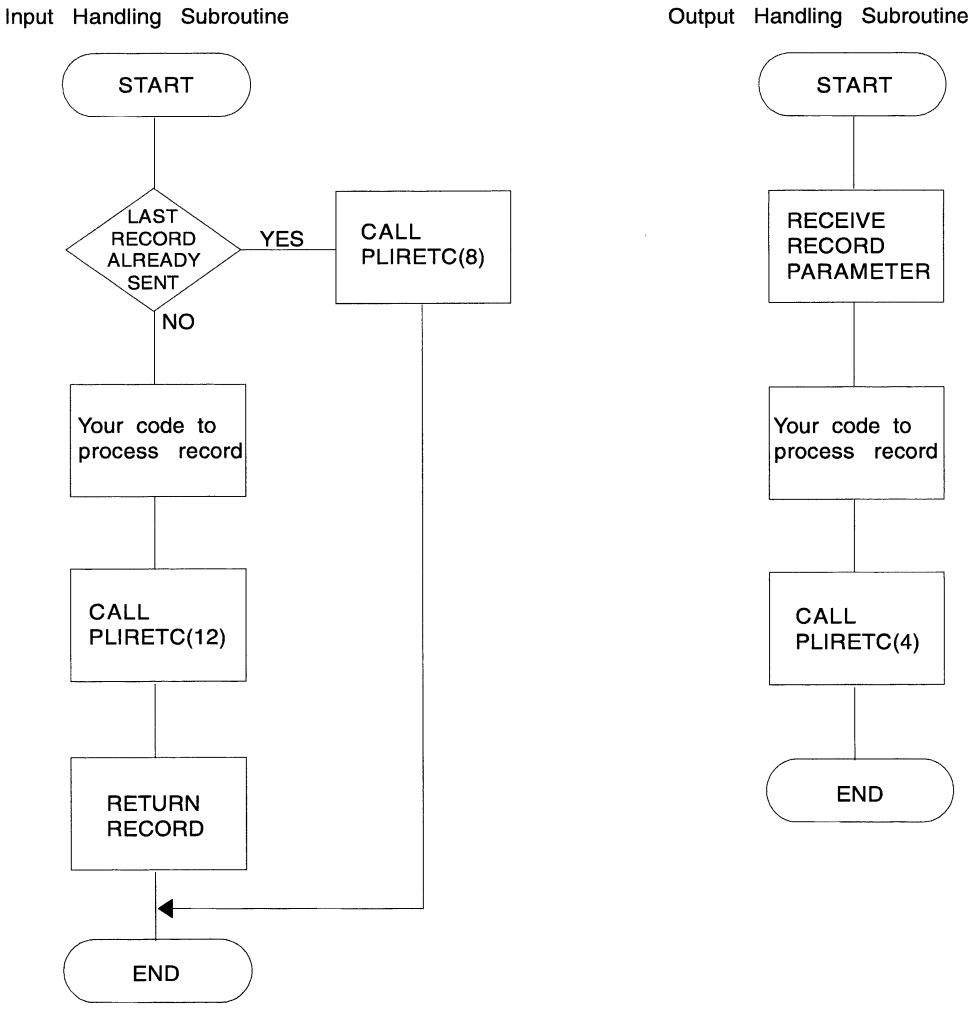


Figure 54. Flowcharts for input and output handling subroutines

Skeletal code for a typical input routine is shown in Figure 55 on page 274.

```
E15: PROC RETURNS (CHAR(80));
    /*---------------------------------------------------------------*/
    /*RETURNS attribute must be used specifying length of data to be */
    /* sorted, maximum length if varying strings are passed to Sort. */
    /*---------------------------------------------------------------*/
  DCL STRING CHAR(80); /*-----------------------------------------*/
                       /*A character string variable 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.                           */
                        /*-----------------------------------------*/
    END;

  ELSE
    DO;
     /*-----------------------------------------------*/
     /* If another record is to be sent to Sort, do the*/
     /* necessary processing, set a return code of 12  */
     /* by calling PLIRETC, and return the data as a   */
     /* character string to Sort                       */
     /*-----------------------------------------------*/

  ****(The code to do your processing goes here)

      CALL PLIRETC (12);  /*-------------------------------------*/
                          /* Set return code of 12, meaning this  */
                          /* record is to be included in the sort */
                          /*-------------------------------------*/
      RETURN (STRING);  /*Return data with RETURN statement*/
    END;

 END;   /*End of the input procedure*/
```

Figure 55. Skeletal code for an input procedure

Examples of an input routine are given in Figure 58 on page 277 and Figure 60 on page 279.

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 PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 59 on page 278.

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 59 on page 278 and Figure 60 on page 279, or to use the sorted data to generate further information. The output handling routine is used by Sort when a call is made to PLISRTC or PLISRTD. When the records have been sorted, Sort passes them, one at a time, to the output handling routine. The output routine then processes them as required. When all the records have been passed, Sort sets up its return code and returns to the statement after the CALL PLISRT*x* statement. There is no indication from Sort to the output handling routine that the last record has been reached. Any end-of-data handling must therefore be done in the procedure that calls PLISRT*x*.

The record is passed from Sort to the output routine as a character string, and you must declare a character string parameter in the output handling subroutine to receive the data. The output handling subroutine must also pass a return code of 4 to Sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

The sort can be stopped by passing a return code of 16 to Sort. This 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 PLISRT*x*, you should declare the parameter with the length of the record. If you specified the record type as V, you should declare the parameter as adjustable, as in the following example:

```
DCL STRING CHAR(*);
```

Figure 61 on page 280 shows a program that sorts varying length records.

A flowchart for a typical output handling routine is given in Figure 54 on page 273. Skeletal code for a typical output handling routine is shown in Figure 56.

```
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 56. 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 PLISRT*x* to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples at the end of this chapter.

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 *LE/VSE Programming Guide.*

```
// JOB    OPT14#7
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 EX106: PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   ' OPTION PRINT=NONE,STORAGE=256K ',
                   RETURN_CODE,
                   ' INPFIL BLKSIZE=80 ',
                   ' OUTFIL BLKSIZE=80 ');
     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);
       OTHERWISE 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;
/*
// EXEC   LNKEDT
// DLBL   SORTOUT,'SORTED.DATA',0
// EXTENT SYS001,VSE222,1,0,3901,1
// ASSGN  SYS001,DISK,VOL=VSE222,SHR
// DLBL   SORTIN1,'UNSORTED.DATA',0
// EXTENT SYS002,VSE222,1,0,3900,1
// ASSGN  SYS002,DISK,VOL=VSE222,SHR
// DLBL   SORTWK1,,0
// EXTENT SYS003,VSE222,1,0,3902,5
// ASSGN  SYS003,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
/&
```

Figure 57. PLISRTA—Sorting from input data set to output data set

Calling PLISRTB example

```
// JOB    OPT14#8
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 EX107:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   ' OPTION PRINT=NONE,STORAGE=256K ',
                   RETURN_CODE,
                   E15X,
                   ' INPFIL EXIT ',
                   ' OUTFIL BLKSIZE=9040 ');
     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);
       OTHERWISE 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 ENV(F MEDIUM(SYSIPT)),
              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;
/*
// EXEC   LNKEDT
// DLBL   SORTOUT,'SORTED.DATA',0
// EXTENT SYS001,VSE222,1,0,3900,1
// ASSGN  SYS001,DISK,VOL=VSE222,SHR
// DLBL   SORTWK1,,0
// EXTENT SYS002,VSE222,1,0,3901,5
// ASSGN  SYS002,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
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 58. PLISRTB—Sorting from input handling routine to output data set

Calling PLISRTC example

```
// JOB    OPT14#9
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 EX108:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   ' OPTION PRINT=NONE,STORAGE=256K ',
                   RETURN_CODE,
                   E35X,
                   ' INPFIL BLKSIZE=80 ',
                   ' OUTFIL EXIT ');
     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);
        OTHERWISE 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;
/*
// EXEC   LNKEDT
// ASSGN  SYS001,SYSIPT
// DLBL   SORTWK1,,0
// EXTENT SYS002,VSE222,1,0,3901,5
// ASSGN  SYS002,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
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 59. PLISRTC—Sorting from input data set to output handling routine

Calling PLISRTD example

```
// JOB    OPT14#10
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
 EX109:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);
     CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   ' OPTION PRINT=NONE,STORAGE=256K ',
                   RETURN_CODE,
                   E15X,
                   E35X,
                   ' INPFIL EXIT ',
                   ' OUTFIL EXIT ');

     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);
       OTHERWISE 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;
/*
// EXEC   LNKEDT
// DLBL   SORTWK1,,0
// EXTENT SYS001,VSE222,1,0,3901,5
// ASSGN  SYS001,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
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 60. PLISRTD—Sorting from input handling routine to output handling routine

Sorting variable-length records example

```
// JOB    OPT14#11
// OPTION LINK
// EXEC   IEL1AA,SIZE=128K
  /* 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*/
                         ' OPTION PRINT=NONE,STORAGE=128K ',
                        RETURN_CODE,
                        PUTIN,
                        PUTOUT,
                        ' INPFIL EXIT ',
                        ' OUTFIL EXIT ');

           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);
              OTHERWISE 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.*/

          PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
          CALL PLIRETC(12);
          RETURN(STRING);
  ENDPUT: END;
```

Figure 61 (Part 1 of 2). Sorting varying-length records using input and output handling routines

```
 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 MAY NOT BE
               DECLARED VARYING*/
         PUT SKIP EDIT(STRING)(A);  /*PRINT THE SORTED DATA*/
         CALL PLIRETC(4);
         END;  /*ENDS PUTOUT*/
     END;
/*
// EXEC   LNKEDT
// DLBL   SORTWK1,,0
// EXTENT SYS001,VSE222,1,0,3905,5
// ASSGN  SYS001,DISK,VOL=VSE222,SHR
// EXEC   ,SIZE=128K
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 61 (Part 2 of 2). Sorting varying-length records using input and output handling routines

Part 6. Specialized programming tasks

Chapter 12. Parameter passing and data descriptors

This chapter describes PL/I parameter passing conventions and also special PL/I control blocks that are passed between PL/I routines at run time. The most important of these control blocks, called locators and descriptors, provide lengths, bounds, and sizes of certain types of argument data.

Assembler routines may communicate with PL/I routines by following the parameter passing techniques described in this chapter. This includes assembler routines that call PL/I routines and PL/I routines that call assembler routines.

For additional information about LE/VSE run-time environment considerations, other than parameter passing conventions, see the *LE/VSE Programming Guide.* This includes run-time environment conventions and assembler macros supporting these conventions.

PL/I parameter passing conventions

PL/I passes arguments using two methods:

- By passing the address of the arguments in the argument list
- By imbedding the arguments in the argument list

This section discusses the first method. For information on the second method, see "Options BYVALUE" on page 289.

When arguments are passed by address between PL/I routines, register 1 points to a list of addresses that is called an argument list. Each address in the argument list occupies a fullword in storage. The last fullword in the list must have its high-order bit turned on for the last argument address to be recognized. If a function reference is used, the address of the returned value or its control block is passed as an implicit last argument. In this situation, the implicit last argument is marked as the last argument, using the high-order bit flagging.

If no arguments are passed in a CALL statement, register 1 is set to zero.

When arguments are passed between PL/I routines, what is passed varies depending upon the type of data passed. The argument list addresses are the addresses of the data items for scalar arithmetic items. For other items, where the receiving routines might expect information about length or format in addition to the data itself, locators and descriptors are used. For program control information such as files or entries, other control blocks are used.

Table 34 on page 285 shows the argument address that is passed between PL/I routines for different data types. The table also includes the effect of the ASSEMBLER option. This option is recommended. See "Passing assembler parameters" on page 285 for additional details.

Table 34. Argument list addresses

Data type passed	Address passed
Arithmetic items	Arithmetic variable
Array or structure	Array or structure variable, if OPTIONS(ASM) Otherwise, aggregate locator[1]
String or area	String or area variable, if OPTIONS(ASM)[2] Otherwise, locator/descriptor[1]
File constant/variable	File variable[3]
Entry	Entry variable[4]
Label	Label variable[5]
Pointer	Pointer variable
Offset	Offset variable

Notes:

1. Locators and descriptors are described below in "Descriptors and locators" on page 291.
2. With options ASSEMBLER: When an unaligned bit string is involved, the address passed points to the byte that contains the start of the bit string. For a VARYING length string, the address passed points to the 2-byte length field that precedes the string.
3. A file variable is a fullword holding the address of file control data.
4. An entry variable consists of two words. The first word has the address of the entry. The second word has the address of the save area for the immediately statically encompassing block or zero, if none.
5. A label variable consists of two words. The first word has the address of a label constant. The second word has the address of the save area of the block that owns the label at the time of assignment. A label constant consists of two words. The first word has the address of the label in the program. The second word has program control data.

Passing assembler parameters

If you call an assembler routine from PL/I, the ASSEMBLER option is recommended in the declaration of the assembler entry name. For example,

```
DCL ASMRTN ENTRY OPTIONS(ASSEMBLER);
DCL C CHAR(80);

CALL ASMRTN(C);
```

When the ASSEMBLER option is specified, the addresses of the data items are passed directly. No PL/I locators or descriptors are involved. In the example above, the address in the argument list is that of the first character in the 80-byte character string.

For details about how argument lists are built, see "PL/I parameter passing conventions" above.

An assembler routine whose entry point has been declared with the ASSEMBLER option can only be invoked by means of a CALL statement. You cannot use it as a function reference. You can avoid the use of function references by passing the returned value field as a formal argument to the assembler routine.

An assembler routine can pass back a return code to a PL/I routine in register 15. If you declare the assembler entry with the option RETCODE, the value passed back in register 15 is saved by the PL/I routine and is accessed when the built-in function PLIRETV is used. If you do not declare the entry with the option RETCODE, any register 15 return code is ignored.

Figure 62 shows the coding for a PL/I routine that invokes an assembler routine with the option RETCODE.

```
P1: PROC;
    DCL A FIXED BIN(31) INIT(3);
    DCL C CHAR(8) INIT('ASM2 RTN');
    DCL AR(3)   FIXED BIN(15);
    DCL ASM2 ENTRY EXTERNAL OPTIONS(ASM RETCODE);
    DCL PLIRETV BUILTIN;

    /* Invoke entry ASM2.                                        */
    /* The argument list has three pointers.                     */
    /* The first  pointer addresses a copy of variable A.        */
    /* The second pointer addresses the storage for C.           */
    /* The third  pointer addresses the storage for AR.          */
    CALL ASM2((A),C,AR);

    /* Check register 15 return code passed back from assembler  */
    /* routine.                                                  */
    IF PLIRETV¬=0 THEN STOP;

    END P1;
```

Figure 62. A PL/I routine that invokes an assembler routine with the option RETCODE

If an assembler routine calls a PL/I procedure, the use of locators and descriptors should be avoided. Although you cannot specify ASSEMBLER as an option in a PROCEDURE statement, locators and descriptors can be avoided if the procedures do not directly receive strings, areas, arrays, or structures as parameters. You could, for example, pass pointers to these items instead. If a length, bound, or size is needed, you can pass these as a separate parameter. If your assembler routine is invoking a PL/I MAIN procedure, see "Passing MAIN procedure parameters" on page 287 for additional considerations involving MAIN procedure parameters.

Figure 63 on page 287 shows a PL/I routine that is invoked by an assembler routine, which is assumed to be operating within the LE/VSE environment.

```
ASMR     CSECT
         .
         .
*
*        Invoke procedure P2 passing four arguments.
*
         LA    1,ALIST            Register 1 has argument list
         L     15,P2              Set P2 entry address
         BALR  14,15              Invoke procedure P2
         .
         .
*
*        Argument list below contains the addresses of 4 arguments.
*
ALIST    DC    A(A)               1st argument address
         DC    A(P)               2nd argument address
         DC    A(Q)               3rd argument address
         DC    A(R+X'80000000')   4th argument address
*
A        DC    F'3'               Fixed bin(31) argument
P        DC    A(C)               Pointer (to C) argument
Q        DC    A(AR)              Pointer (to AR) argument
R        DC    A(ST)              Pointer (to ST) argument
*
C        DC    CL10'INVOKE P2 '    Character string
AR       DC    4D'0'              Array of 4 elements
ST       DC    F'1'               Structure
         DC    F'2'
         DC    D'0'
P2       DC    V(P2)              Procedure P2 entry address
         END   ASMR
```

```
/* This routine receives four parameters from ASMR.     */
/* The arithmetic item is received directly.            */
/* The character string, array and structure are        */
/* received using a pointer indirection.                */

P2: PROC(A,P,Q,R);

    DCL A FIXED BIN(31);
    DCL (P,Q,R) POINTER;
    DCL C CHAR(10) BASED(P);
    DCL AR(4)   FLOAT DEC(16) BASED(Q);
    DCL 1 ST BASED(R),
          2 ST1 FIXED BIN(31),
          2 ST2 FIXED BIN(31),
          2 ST3 FLOAT DEC(16);
          .
          .
          .
    END P2;
```

Figure 63. A PL/I routine that is invoked by an assembler routine

If you choose to code an assembler routine that passes or receives strings, areas, arrays or structures that require a locator or descriptor, see "Descriptors and locators" on page 291 for their format and construction. Keep in mind, however, that doing so may affect the migration of these assembler routines in the future.

Passing MAIN procedure parameters

The format of a PL/I MAIN procedure parameter list is controlled by the SYSTEM compiler option and the NOEXECOPS procedure option. The kind of coding needed also depends upon the type of parameter received by the MAIN procedure.

If the MAIN procedure receives no parameters or a single varying character string, then:

- It is recommended that the MAIN procedure be compiled with SYSTEM(VSE).

- If the assembler routine needs to pass run-time options for initializing the run-time environment, then the NOEXECOPS option should not be specified or defaulted. Otherwise, NOEXECOPS should be specified.
- The MAIN procedure must be coded to have no parameters or a single parameter, consisting of a varying character string. For example:

```
MAIN: PROC(C) OPTIONS(MAIN);
      DCL C CHAR(100) VARYING;
```

- The assembler routine must invoke the MAIN procedure passing a varying length character string, as shown in Figure 64. The string has the same format as that passed in the PARM= option of the VSE EXEC JCL statement (see "Program run-time considerations" on page 59). The string consists of optional run-time options, followed by a slash (/), followed by optional characters to be passed to the MAIN procedure.

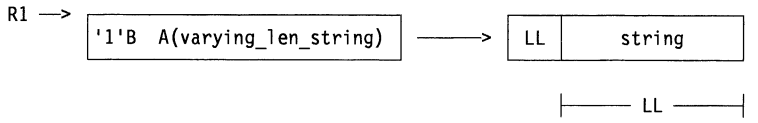


Figure 64. Assembler routine invoking a MAIN procedure

If NOEXECOPS is specified, run-time options and the accompanying slash should be omitted. If run-time options are provided in the string, they will have no effect on environment initialization.

If a MAIN procedure receives no parameters, the argument list should be built as shown in Figure 64, but a null string should be passed by setting the length LL to zero.

If the MAIN procedure receives a character string as a parameter, the locator/descriptor needed for this string is built by PL/I run-time services before the MAIN procedure gains control.

If a MAIN procedure receives more than one parameter or a parameter that is not a single varying character string, then:

- It is recommended that the MAIN procedure be compiled with SYSTEM(VSE).
- NOEXECOPS is always implied. There is no mechanism to receive and parse run-time options from the assembler routine for initializing the run-time environment.
- The assembler routine should build its argument list so it corresponds to the parameters expected by the MAIN procedure. The assembler argument list is passed through as is to the MAIN procedure. If strings, areas, arrays or structures need to be passed, consider passing pointers to these items instead. This avoids the use of locators and descriptors.

  Figure 65 on page 289 illustrates this technique.

```
ASM0     CSECT
         .
         .
         .
*
*        Invoke MAINP passing two arguments.
*
         CDLOAD MAINP              Load phase MAINP
         LR    15,1                Entry point (CEESTART) in R15
         LA    1,ALIST             Register 1 has argument list
         BALR  14,15               Invoke MAINP program,
*                                  giving control to entry CEESTART
         .
         .
         .
*
*        The argument list below contains the addresses of two
*        arguments, both of which are pointers.
*
ALIST    DC    A(P)                First argument address
         DC    A(Q+X'80000000')    Second argument address
*
P        DC    A(C)                Pointer (to 1st string) argument
Q        DC    A(D)                Pointer (to 2nd string) argument
*
C        DC    C'Character string 1  '
D        DC    C'Character string 2  '
         END   ASM0
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

%PROCESS SYSTEM(VSE);

 /* This procedure receives two pointers as its parameters.       */

 MAINP: PROCEDURE(P,Q) OPTIONS(MAIN);
       DCL (P,Q) POINTER;
       DCL C CHAR(20) BASED(P);
       DCL D CHAR(20) BASED(Q);

       /* Display contents of character strings pointed to by     */
       /* pointer parameters.                                     */
       DISPLAY(C);
       DISPLAY(D);

       END MAINP;
```

Figure 65. Assembler routine that passes pointers to strings

- An assembler routine may choose to pass strings, areas, arrays or structures to a MAIN procedure. If so, locators and descriptors as described in "Descriptors and locators" on page 291 must be provided by the assembler routine.

Assembler routines should not invoke MAIN procedures compiled by PL/I VSE that specify SYSTEM(CICS). SYSTEM(VSE) should be used even if the parameter to the MAIN procedure is not a single varying-length character string. If the program contains DL/I calls, SYSTEM(DLI) or SYSTEM(DL1) may be specified.

Options BYVALUE

PL/I supports the BYVALUE option for external procedure entries and entry declarations. The BYVALUE option specifies that arguments are passed by copying the value of the arguments into the argument list. This implies that the invoked routine can not modify the arguments passed by the caller.

BYVALUE arguments and returned values must have a scalar data type of either POINTER or REAL FIXED BINARY(31,0). Consequently, each fullword slot in the

argument list consists of either a pointer value or a REAL FIXED BINARY(31,0) value. If you need to pass a data type that is not POINTER or REAL FIXED BINARY(31,0), consider passing the data type indirectly using the POINTER data type.

Values from function references are returned using register 15.

Figure 66 shows an assembler routine ASM1 that invokes a procedure PX passing three arguments BYVALUE, which in turn invokes an assembler routine ASM2 (not shown). The assembler routines are assumed to be operating in the LE/VSE environment.

```
ASM1     CSECT
         .
         .
         .
*
*        Invoke procedure PX passing three arguments BYVALUE.
*
         LA    1,ALIST            Register 1 has argument list
         L     15,VPX             Set reg 15 with entry point
         BALR  14,15              Invoke BYVALUE procedure
         LTR   15,15              Check returned value
         .
         .
*
*        The BYVALUE argument list contains three arguments.
*        The high order bit of the last argument is *not* specially
*        flagged when using the BYVALUE passing convention.
*
ALIST    DC    A(A)               1st arg is pointer value
         DC    A(C)               2nd arg is pointer value
         DC    F'2'               3rd arg is arithmetic value
*
A        DC    2D'0'              Array
C        DC    C'CHARSTRING'      Character string
*
*
VPX      DC    V(PX)              Entry point to be invoked
         END   ASM1

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

PX:   PROCEDURE(P,Q,M) OPTIONS(BYVALUE) RETURNS(FIXED BIN(31));
      DCL (P,Q) POINTER;
      DCL A(2) FLOAT DEC(16) BASED(P);
      DCL C    CHAR(10)      BASED(Q);
      DCL M FIXED BIN(31);
      DCL  ASM2 ENTRY(FIXED BIN(31,0)) OPTIONS(BYVALUE ASM);

      M=A(1);

      /* ASM2 is passed variable M byvalue, so it can not alter  */
      /* the contents of variable M.                             */
      CALL ASM2(M);

      RETURN(0);

      END PX;
```

Figure 66. Assembler routine passing arguments BYVALUE

Descriptors and locators

PL/I supports locators and descriptors in order to communicate the lengths, bounds, and sizes of strings, areas, arrays and structures between PL/I routines. For example, the procedure below can receive the parameters shown without explicit knowledge of their lengths, bounds or sizes at compilation time.

```
P:  PROCEDURE(C,D,AR);
    DCL C  CHAR(*),
        D  AREA(*),
        AR(*,*) CHAR(*);
```

The called procedure P can use PL/I built-in functions such as LBOUND, HBOUND, and LENGTH to determine the dimensions and sizes of the parameters at run time.

The use of locators and descriptors is not recommended for assembler routines, because the migration of these routines may be affected in the future. In addition, portability to other platforms can be adversely affected. See "Passing assembler parameters" on page 285 for techniques to avoid the use of these control blocks.

The major control blocks are:

Descriptors These hold the extent of the data item such as string lengths, array bounds, and area sizes.

Locators These hold the address of a data item. If they are not concatenated with the descriptor, they hold the descriptor's address.

Locator/descriptor
This is a control block consisting of a locator concatenated with a descriptor.

When received as parameters or passed as arguments, locators and descriptors are needed for strings, areas, arrays, and structures. For strings and areas, the locator is concatenated with the descriptor and contains only the address of the variable. For structures and arrays, the locator is a separate control block from the descriptor and holds the address of both the variable and the descriptor.

Figure 67 on page 292 gives an example of the way in which data is related to its locator and descriptor.

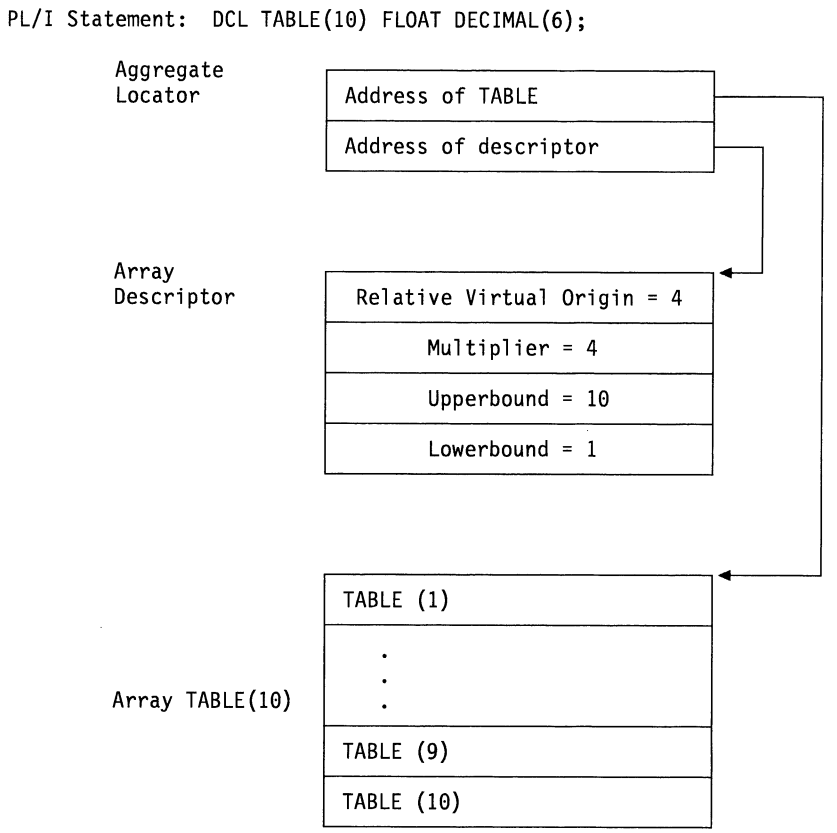


Figure 67. Example of locator, descriptor, and array storage

Aggregate locator

The aggregate locator holds this information:

- The address of the start of the array or structure
- The address of the array descriptor or structure descriptor

Figure 68 shows the format of the aggregate locator.

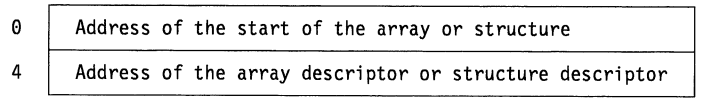


Figure 68. Format of the aggregate locator

The array and structure descriptor are described in subsequent sections.

Area locator/descriptor

The area locator/descriptor holds this information:

- The address of the start of the area
- The length of the area

Figure 69 on page 293 shows the format of the area locator/descriptor.

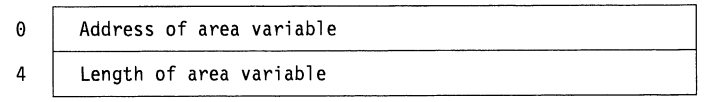


Figure 69. Format of the area locator/descriptor

The area variable consists of a 16-byte area variable control block followed by the storage for the area variable.

The area descriptor is the second word of the area locator/descriptor. It is used in structure descriptors when areas appear in structures and also in array descriptors, for arrays of areas.

Array descriptor

The array descriptor holds this information:

- The relative virtual origin (RVO) of the array
- The multiplier for each dimension
- The high and low bounds for the subscripts in each dimension

When the array is an array of strings or areas, the string or area descriptor is concatenated at the end of the array descriptor to provide the necessary additional information. String and area descriptors are the second word of the locator/descriptor word pair.

Figure 70 shows the format of the array descriptor.

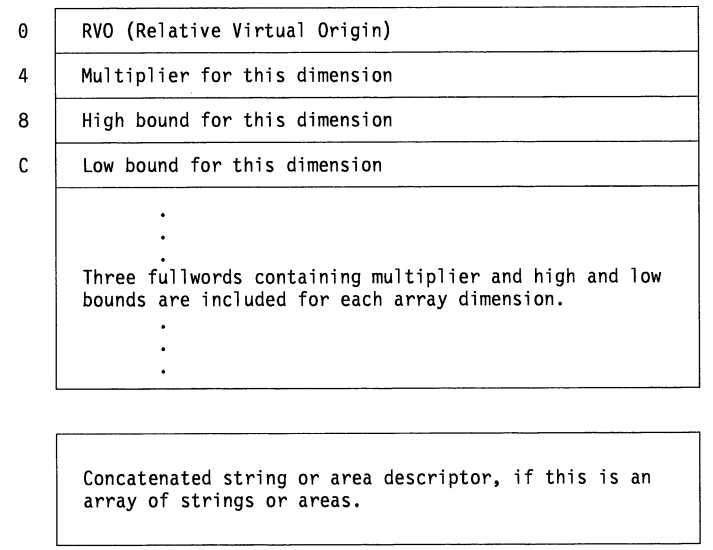


Figure 70. Format of the array descriptor

RVO (Relative virtual origin)
This is held as a byte value except for bit string arrays, in which case this is a bit value. For bit string arrays, the bit offset from the byte address is held in the string descriptor.

Multiplier The multiplier is held as a byte value, except for bit string arrays in which case they are bit values.

High bound The high subscript for this dimension

Low bound The low subscript for this dimension.

String locator/descriptor

The string locator/descriptor holds this information:

- The byte address of the string
- The (maximum) length of the string
- Whether or not it is a varying string
- For a bit string, the bit offset from the byte address

Figure 71 shows the format of the string locator/descriptor.

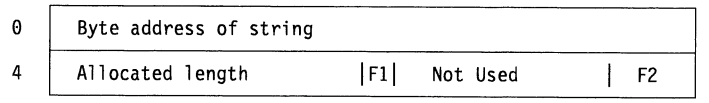


Figure 71. Format of the string locator/descriptor

For VARYING strings, the byte address of the string points to the half-word length prefix, which precedes the string.

Allocated length is held in bits for bit strings and in bytes for character strings. Length is held in number of graphics for graphic strings.

```
F1 = First bit in second halfword:
     '0'B  Fixed length string
     '1'B  VARYING string

F2 = Last 3 bits in second halfword:
     Used for bit strings to hold offset from byte address of
     first bit in bit string.
```

The string descriptor is the second word of the string locator/descriptor. It is used in structure descriptors when strings appear in structures and also in array descriptors, for arrays of strings.

Structure descriptor

The structure descriptor is a series of fullwords that give the byte offset of the start of each base element from the start of the structure. If a base element has a descriptor, the descriptor is included in the structure descriptor, following the appropriate fullword offset.

Structure descriptor format

Figure 72 on page 295 shows the format of the structure descriptor. For each base element in the structure, a fullword field is present in the structure descriptor that contains the byte offset of the base element from the start of the storage for the structure. If the base element is a string, area, or array, this fullword is followed by a descriptor, which is followed by the offset field for the next base element. If the base element is not a string, area, or array, the descriptor field is omitted.

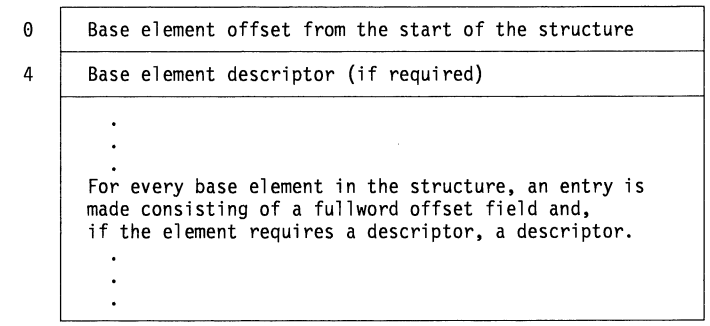


Figure 72. Format of the structure descriptor

Base element offsets are held in bytes. Any adjustments needed for bit-aligned addresses are held in the respective descriptors.

Major and minor structures themselves, versus the contained base elements, are not represented in the structure descriptor.

Arrays of structures and structures of arrays

When necessary, an aggregate locator and a structure descriptor are generated for both arrays of structures and structures of arrays.

The structure descriptor has the same format for both an array of structures and a structure of arrays. The difference between the two is the values in the field of the array descriptor within the structure descriptor. For example, take the array of structures AR and the structure of arrays ST, declared as follows:

Array of structures
```
DCL 1 AR(10),
      2 B,
      2 C;
```

Structure of arrays
```
DCL 1 ST,
      2 B(10),
      2 C(10);
```

The structure descriptor for both AR and ST contains an offset field and an array descriptor for B and C. However, the values in the descriptors are different, because the array of structures AR consists of elements held in the order:

```
B,C,B,C,B,C,B,C,B,C,B,C,B,C,B,C,B,C,B,C
```

but the elements in the structure of arrays ST are held in the order:

```
B,B,B,B,B,B,B,B,B,B,C,C,C,C,C,C,C,C,C,C.
```

Chapter 13. 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.

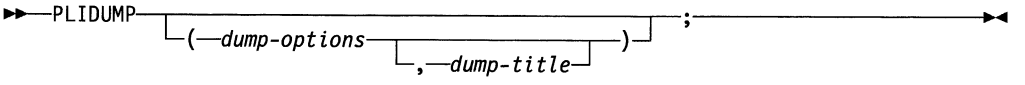
Note: PL/I uses the dump services of IBM Language Environment for VSE/ESA to produce your dump. The compiler constructs a parameter list for CEE5DMP based on your parameters to PLIDUMP, and calls CEE5DMP to produce the dump. You can call either PLIDUMP or CEE5DMP to obtain a dump of your program's storage. PLIDUMP is described here, and CEE5DMP is described in the *LE/VSE Programming Guide* book.

Figure 73 shows an example of a PL/I routine calling PLIDUMP to produce an LE/VSE 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 73. Example PL/I routine calling PLIDUMP

The syntax and options for PLIDUMP are shown below.



dump-options
is a character string consisting of one or more of the following:

B	BLOCKS. To produce a hexadecimal dump of PL/I control blocks.
NB	NOBLOCKS.
C	Continue. The routine continues after the dump.
E	Exit. The enclave is terminated with a dump.
F	FILES. To request a set of attributes for all open files, and the contents of their buffers.
NF	NOFILES.
H	STORAGE. To request a hexadecimal dump of the main storage partition used by the dump.
NH	NOSTORAGE.
K	BLOCKS (when running under CICS). The Transaction Work Area is included.
NK	NOBLOCKS (when running under CICS).
S	STOP. The enclave is terminated with a dump.
T	TRACEBACK. Requests a trace of active procedures, begin-blocks, on-units, and library modules.
NT	NOTRACEBACK.

The defaults for these options are T, F, and C.

In addition, the following are accepted for compatibility with the DOS PL/I Optimizing Compiler. If specified, they will be ignored.

D	File analysis.
ND	No file analysis.
Q	Use the DOS system dump routines to produce the dump.
NQ	Do not use the DOS dump routines.
R	Report on current storage usage.
NR	No storage report.
48	Use the PL/I 48-character set.
60	Use the PL/I 60-character set.

dump-title
is a user-identified character string up to 80 characters long that is printed as the dump header. If your program calls PLIDUMP a number of times, you should use a different dump header each time. This simplifies identifying the beginning of each dump.

PLIDUMP output destination

The output from PLIDUMP will be directed to the following destinations:

STORAGE dump output

Output for STORAGE dumps (option H) will be directed to a dump sublibrary, provided that the JCL OPTION SYSDUMP is in effect and a dump sublibrary has been defined for the partition with a LIBDEF DUMP command. Otherwise the output will go to SYSLST.

Other dump output

For all the other PLIDUMP options, if there is a DLBL or TLBL statement in the JCL for PLIDUMP, PL1DUMP, or CEEDUMP, the dump output will be directed to that data set. Otherwise the output will go to SYSLST.

Chapter 14. 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.

Note: You cannot use Checkpoint/Restart in a dynamic partition, or in a partition that extends beyond the 16-megabyte line.

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.

The restart is performed later as a new job.

PL/I Checkpoint/Restart uses the Checkpoint/Restart Facility of the operating system. This facility is described in the books listed in "Bibliography" on page 343.

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 records can be written.

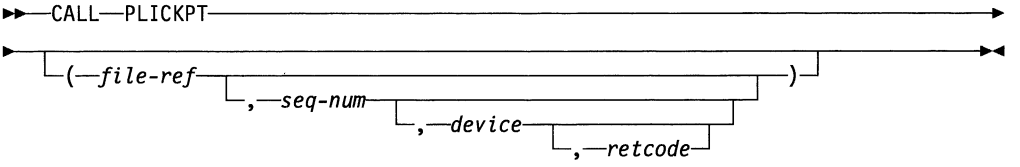
Note: You should be aware of the restrictions affecting data sets used by your program. For example:

- All VSAM files must be closed before calling PLICKPT,
- Sequential files that are supported by VSE/VSAM Space Management for SAM cannot be repositioned by the restart program.

These restrictions are documented in the *VSE/ESA System Macros Reference* book.

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.



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.

file-ref
is a character string constant or variable specifying the name of the file defining the data set that is to be used for checkpoint records. If you omit this argument and the data set is on a disk device, the system will use the default name SYSCHK. This name must be included in a DLBL statement in your JCL, to identify the checkpoint data set.

For checkpoint data sets on magnetic tape, *file-ref* must be omitted.

seq-num
is a character string variable which the system will use to record the sequence number of the checkpoint record so that you can identify it later. Your program should print this sequence number as soon as each checkpoint has been taken, so that, if a restart is necessary, the latest checkpoint can be identified.

The *seq-num* variable should be a fixed-length character string with a length of 4 or more. The sequence number is four bytes long, and will be stored in the first four bytes of the variable.

device
is a character string constant or variable that defines the device used for the checkpoint data set. It must be of the form 'SYS*nnn*', where *nnn* is between 000 and 255. If you omit this argument, a default of SYS001 will be used.

Note: For compatibility with the DOS PL/I Optimizing Compiler, you can also specify a device type, for example:

```
'SYS001,3350'
```

The compiler will ignore the device type specification.

retcode
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 JCL 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.

Defining the checkpoint data set

The data set that holds the checkpoint records can be on disk or tape, and it must have CONSECUTIVE organization. Its location is determined by the parameters passed to the PLICKPT subroutine, and by the JCL used to run the program.

If the run-time JCL contains a DLBL statement that matches the *file-ref* parameter (either specified or defaulted), then that disk data set will be used for the checkpoint records. The device specified on the EXTENT statement overrides the device specified or defaulted in the *device* parameter.

If there is no DLBL statement to match the *file-ref* parameter, the checkpoint records will be written to magnetic tape, and a tape device must be assigned to the logical unit number specified or defaulted in the *device* parameter.

Taking checkpoints on magnetic tape

In the following example, SYS001 is assigned to a magnetic tape device, so that device is used for the checkpoint records.

```
CALL PLICKPT ('', SEQUENCE_NUM, 'SYS001', RETURN_CODE);
```

Checkpoint data sets on magnetic tape are treated as unlabeled tape data sets, so there should not be a TLBL statement in the JCL. If you use a labeled magnetic tape exclusively for recording checkpoint records, you should include the MTC job control statement to position the tape past the label.

Any number of checkpoints can be recorded consecutively on magnetic tape. A separate magnetic-tape device can be used exclusively for the checkpoints, or, if there are no additional devices available, it is possible to include checkpoint records among the records of another data set being written onto magnetic tape. The checkpoint records will be ignored by VSE data management when the data set is accessed for input by a PL/I program (except when the data set is accessed by a SEQUENTIAL UNBUFFERED file or an ASCII file). Similarly, records that are not checkpoint records will be ignored by the VSE restart routines when the data set is used to restart a program.

Taking checkpoints on disk storage

Disk storage will be used for the checkpoint data set when there is a DLBL JCL statement that matches the *file-ref* parameter. For example:

```
CALL PLICKPT ('SYSCHK', SEQUENCE_NUM, '', RETURN_CODE);
```

In this example, there must be a DLBL statement for SYSCHK that defines the checkpoint data set.

You can record any number of checkpoint records on a disk storage volume. However, the number of checkpoint records that will be present on the volume at any one time depends on the amount of space made available to the checkpoint data set. When writing checkpoint records, if the extent is full, the system will restart at the beginning of the extent and overwrite earlier checkpoint records.

You can estimate the amount of space needed for the data set as follows:

```
space = (1+CEIL(P/T))*R
```

where

space is the amount of space required in tracks for CKD or ECKD devices, or in blocks for FBA devices,

P is the partition size that is specified on the EXEC statement used to run the PL/I program,

T is the capacity in bytes of the DASD device used. This is the track capacity for CKD devices, or the hardware-dependent blocksize for fixed-block devices,

R is the number of checkpoint records to be retained.

Requesting a restart

To restart a program after a failure, you need to use the RSTRT job control statement. The format of this statement is

```
// RSTRT SYSnnn,rrrr[,filename]
```

where

SYSnnn
is the logical unit name of the device on which the checkpoint file is stored,

rrrr
is the sequence number of the checkpoint record to be used for restarting,

filename
is the name of the disk checkpoint file that was used for the program's original run.

The RSTRT statement takes the place of the EXEC statement in your JCL. You can request a restart of an interrupted job step by using the original job step's JCL, replacing the step's EXEC statement with a RSTRT statement.

You can find a full description of the RSTRT statement in the *VSE/ESA System Control Statements* book, and the *VSE/ESA Guide to System Functions.*

Note: During a restart, PL/I will re-establish the setting of the LE/VSE TRAP run-time option that was in effect for the checkpointed program. If the program was running with TRAP(OFF), and it had established its own STXIT routines, then TRAP(OFF) will be re-established but the STXIT routines will not.

Automatic restart within a program

Automatic restart functions are not available under VSE. However, for compatibility with the MVS implementation of PL/I, you can code a call to the built-in subroutine PLIREST in your program. This call should be of the following form:

```
CALL PLIREST;
```

When the system encounters this statement, it will issue a message on the system console indicating that automatic restart is not available, and ABEND the program. This will then allow you to perform a manual restart using the RSTRT JCL statement.

Modifying checkpoint/restart activity

In some implementations, it is possible to cancel any automatic restart activity from checkpoints taken in your program by calling the built-in subroutine PLICANC, as:

```
CALL PLICANC;
```

However, VSE does not support this function. If your program calls PLICANC, the system will issue a message and the program will continue normal operation.

Part 7. Appendix

Appendix. Sample program IEL1ESO1

This appendix is a PL/I program that illustrates all the components of the listings produced by the compiler and the linkage editor. You can use this sample program to verify that PL/I has been installed correctly on your system.

The listings themselves are described in the chapters on compiling.

The program has comments to document both the preprocessor input and the source listing. These comments are the lines of text preceded by /* and followed by */. Note that the /* does not appear in columns 1 and 2 of the input record, because /* in those columns is understood as a job control end-of-file statement.

In addition to the /* comments lines, most pages of the listing contain brief notes explaining the contents of the pages.

```
5686-069 IBM PL/I for VSE/ESA          Ver 1 Rel 1 Mod 0                    27 OCT 94   13:04:31     PAGE    1
OPTIONS SPECIFIED
%PROCESS OPTIONS INSOURCE SOURCE NEST MACRO MAP STORAGE;                                00010000
%PROCESS AGGREGATE, ESD, OFFSET;                                                       00015000
%PROCESS LIST(40,45) FLAG(I) MARGINS(2,72,1) MARGINI('|');                             00020000
%PROCESS OPT(2) TEST(ALL,SYM) ATTRIBUTES(FULL) XREF(SHORT);                            00030000
OPTIONS USED

AGGREGATE       NOGONUMBER      ATTRIBUTES(SHORT)
ESD             NOGRAPHIC       NOCOMPILE(S)
GOSTMT          NOINCLUDE       CMPAT(V2)
INSOURCE        NOMDECK         FLAG(I)
LMESSAGE        NONUMBER        LANGLVL(OS,NOSPROG)
MACRO                           LINECOUNT(55)
MAP                             LIST(40,45)
NEST                            MARGINI('|')
OFFSET                          MARGINS(2,72,1)
OPTIONS                         OPTIMIZE(TIME)
SOURCE                          SEQUENCE(73,80)
STMT                            SIZE(7925760)
STORAGE                         NOSYNTAX(S)
                                SYSTEM(VSE)
                                TEST(ALL,SYM)
                                XREF(SHORT)
```

1 2

Start of the compiler listing

1 List of options specified in %PROCESS statements.

2 List of options used, whether obtained by default or by being specified explicitly.

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE   2
                  PREPROCESSOR INPUT
 2                                                                                                      3
LINE
   1       |/* PL/I Sample Program: Used to verify product installation  1     */|00040000

   3       |/*=== SAMPLE ============================================================*/|00060000
   4       |/*=========================================================================*/|00070000
   5       |/*==                                                                    ==*/|00080000
   6       |/*==  This is the PL/I sample program that is intended to be           ==*/|00090000
   7       |/*==  used to verify the product's complete installation.              ==*/|00100000
   8       |/*==  It is expected to execute and to provide some output.            ==*/|00110000
   9       |/*==  Although "results" are created by the program it is only        ==*/|00120000
  10       |/*==  to verify that representative I/O services are operable --      ==*/|00130000
  11       |/*==  the results are verified (internally) by the program.           ==*/|00140000
  12       |/*==                                                                    ==*/|00150000
  13       |/*==  The program is intended to read a data file and count           ==*/|00160000
  14       |/*==  the number of occurrences of each PL/I statement type.          ==*/|00170000
  15       |/*==  The results are displayed at the end of execution.              ==*/|00180000
  16       |/*==  If any count does not match the value that is expected         ==*/|00190000
  17       |/*==  a warning message is displayed.                                 ==*/|00200000
  18       |/*==                                                                    ==*/|00210000
  19       |/*==  When the program is executed this source program file will      ==*/|00220000
  20       |/*==  be used as the input file.  The filename or DDNAME is           ==*/|00230000
  21       |/*==  SOURCE.                                                         ==*/|00240000
  22       |/*==                                                                    ==*/|00250000
  23       |/*==  NOTE: Compilation of this program should cause preprocessor     ==*/|00260000
  24       |/*==        message:                                                  ==*/|00270000
  25       |/*==                                                                    ==*/|00280000
  26       |/*==   IEL2250I I 140 The WORD_TABLE was successfully declared.      ==*/|00281000
  27       |/*==                                                                    ==*/|00282000
  28       |/*==        Two compiler messages will be produced as well:           ==*/|00283000
  29       |/*==                                                                    ==*/|00284000
  30       |/*==   IEL0533I I     NO 'DECLARE' STATEMENT(S) FOR 'INDEX'.          ==*/|00285000
  31       |/*==   IEL0871I I  62 RESULT OF BUILTIN FUNCTION 'SUM' WILL BE       ==*/|00286000
  32       |/*==                   EVALUATED USING FIXED POINT ARITHMETIC        ==*/|00287000
  33       |/*==                   OPERATIONS.                                   ==*/|00288000
  34       |/*==                                                                    ==*/|00289000
  35       |/*=========================================================================*/|00290000
  36       |/*=========================================================================*/|00300000

  38       |SAMPLE: PROCEDURE OPTIONS(MAIN) REORDER;                                |00320000
```

Source statements for the sample program as they appear in the input stream. These statements form the input data for the preprocessor. Preprocessor statements are identified by the % symbol.

1 The first line of the input is included as part of the heading for all the pages of the preprocessor and compiler listing.

2 Each input record is numbered sequentially.

3 If an input record has a sequence number, this number is printed.

```
 LINE
   40        %/*-------------------------------------------------------------*/;|00340000
   41        %/*   The services of the PL/I Preprocessor will be used by this  */;|00350000
   42        %/* program.  Since some of its variables are global (their use   */;|00360000
   43        %/* crosses macros) they must be defined early in the source      */;|00370000
   44        %/* program.                                                      */;|00380000
   45        %/*   Notice that these lines start with a percent sign and       */;|00390000
   46        %/* end with a semicolon.  Notice also that they do not appear    */;|00400000
   47        %/* on the program SOURCE listing.                                */;|00410000
   48        %/*-------------------------------------------------------------*/;|00420000
   49        %DCL BIG_LIST CHAR;                                                 |00430000
   50        %BIG_LIST = '';                                                     |00440000
   51        %;                                                                  |00450000
   52        %DCL SIZE_WORD_LIST FIXED;                                          |00460000
   53        %SIZE_WORD_LIST = 0;                                                |00470000
   54        %;                                                                  |00480000
   55        %DCL MAX_WORD_LENGTH FIXED;                                         |00490000
   56        %MAX_WORD_LENGTH = 0;                                               |00500000
   57        %;                                                                  |00510000
   58        %DCL CURRENT_POSITION FIXED;                                        |00520000
   59        %CURRENT_POSITION = 0;                                              |00530000
   60        %;                                                                  |00540000
   61        %DCL FIRST_WORD_INDICES CHAR;                                       |00550000
   62        %FIRST_WORD_INDICES = '';                                           |00560000
   63        %DCL LAST_INDEX         FIXED;                                      |00570000
   64        %LAST_INDEX = 0;                                                    |00580000
   65        %;                                                                  |00590000
   66        %ACTIVATE ADD_TO_LIST, END_OF_LIST;                                 |00600000
   67        %/*-------------------------------------------------------------*/;|00610000
   68        %/* End of the global Preprocessor variable declarations        */;|00620000
   69        %/*-------------------------------------------------------------*/;|00630000
   70        /*=============================================================*/|00640000
   71        /* Non-Preprocessor data variables are declared here.  Only the  */|00650000
   72        /* variables that are used in the main block (or in more than one */|00660000
   73        /* of the contained blocks) are defined here.                    */|00670000
   74        /*=============================================================*/|00680000
   75                                                                            |00690000
   76        /*-------------------------------------------------------------*/|00700000
   77        /* Declare the source program input file and its accoutrements.  */|00710000
   78        /*-------------------------------------------------------------*/|00720000
   79        DECLARE SOURCE         FILE RECORD ENVIRONMENT(F RECSIZE(80));      |00730000
   80        DECLARE RECORD         CHARACTER(80);                               |00740000
   81        DECLARE RECORD_READ    BIT(1) INIT(FALSE);                          |00750000
   82        DECLARE LAST_CHAR_POSN FIXED BINARY(15);                            |00760000
   83        DECLARE DISCREPANCY_OCCURRED  BIT(1) INIT(FALSE);                   |00770000
   84                                                                            |00780000
   85        /*-------------------------------------------------------------*/|00790000
   86        /* Declare the left- and right-margins of the input dataset.     */|00800000
   87        /*-------------------------------------------------------------*/|00810000
   88        DECLARE LEFT_MARGIN    FIXED BINARY(15) INIT('2');                  |00820000
   89        DECLARE RIGHT_MARGIN   FIXED BINARY(15) INIT('72');                 |00830000
   90                                                                            |00840000
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE    4
 LINE
   91        |  /*-------------------------------------------------------------------*/|00850000
   92        |  /* Declare '1'B as TRUE and '0'B as FALSE.                          */|00860000
   93        |  /*-------------------------------------------------------------------*/|00870000
   94        |  DECLARE TRUE           BIT(1) INIT('1'B);                            |00880000
   95        |  DECLARE FALSE          BIT(1) INIT('0'B);                            |00890000
   96        |                                                                       |00900000
   97        |  /*-------------------------------------------------------------------*/|00910000
   98        |  /* Declare which characters are acceptable as the first character  */|00920000
   99        |  /* of a word -- then declare acceptable succeeding characters.     */|00930000
  100        |  /*-------------------------------------------------------------------*/|00940000
  101        |  DECLARE WORD_FIRST_CHARACTERS CHAR(29) STATIC                        |00950000
  102        |                INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ@#$');                 |00960000
  103        |  DECLARE WORD_NEXT_CHARACTERS  CHAR(30) STATIC                        |00970000
  104        |                INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ_@#$');                |00980000
  105        |                                                                       |00990000
  106        |  /*-------------------------------------------------------------------*/|01000000
  107        |  /* Declare a place to hold words extracted from program text.      */|01010000
  108        |  /*-------------------------------------------------------------------*/|01020000
  109        |  DECLARE WORD  CHAR(31) VARYING;                                      |01030000
  110        |  DECLARE WORD_INDEX     FIXED BINARY(15);                             |01040000
  111        |                                                                       |01050000
  112        |  /*-------------------------------------------------------------------*/|01060000
  113        |  /* Declare the use of SYSPRINT and all of the builtin functions.   */|01070000
  114        |  /*-------------------------------------------------------------------*/|01080000
  115        |  DECLARE SYSPRINT       FILE STREAM OUTPUT PRINT ENV(MEDIUM(SYSLST)),  |01090000
  116        |         PLIXOPT CHAR(100) VAR STATIC EXT INIT('MSGFILE(SYSPRINT)');   |01095000
  117        |  DECLARE (HIGH, SUBSTR, SUM, UNSPEC, VERIFY) BUILTIN;                 |01100000
  118        |  DECLARE ONCODE         BUILTIN;                                      |01110000
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE    5
 LINE
  119        |  /*===================================================================*/|01120000
  120        |  /* PL/I statement keywords are collected using the ADD_TO_LIST     */|01130000
  121        |  /* macro.  They are put into a table, WORD_TABLE, by the           */|01140000
  122        |  /* END_OF_LIST macro.  That macro also creates an index,           */|01150000
  123        |  /* WORD_TABLE_INDEX, into the WORD_TABLE.                          */|01160000
  124        |  /*                                                                 */|01170000
  125        |  /* Finally, a table, WORD_COUNT, is created that has a counter     */|01180000
  126        |  /* that corresponds to each word.  Whenever that word is           */|01190000
  127        |  /* encountered in the input stream the appropriate WORD_COUNT      */|01200000
  128        |  /* element is incremented.                                         */|01210000
  129        |  /*                                                                 */|01220000
  130        |  /* Notice that there are no semicolons in the macro statements.    */|01230000
  131        |  /*===================================================================*/|01240000
  132        |  ADD_TO_LIST ('ALLOCATE,BEGIN')                                       |01260000
  133        |  ADD_TO_LIST ('CALL,CLOSE,DCL,DECLARE,DEFAULT,DISPLAY')               |01270000
  134        |  ADD_TO_LIST ('DO')                                                   |01280000
  135        |  ADD_TO_LIST ('ELSE,END,ENTRY,FREE,GENERIC,GET,GO,GOTO,IF')           |01290000
  136        |  ADD_TO_LIST ('LEAVE,LIST,LOCATE,ON,OPEN')                            |01300000
  137        |  ADD_TO_LIST ('PROC,PROCEDURE')                                       |01310000
  138        |  ADD_TO_LIST ('READ,RETURN,REVERT,REWRITE,SELECT,SIGNAL')             |01320000
  139        |  ADD_TO_LIST ('STOP,THEN,WAIT,WHEN,WRITE')                            |01330000
  140        |  END_OF_LIST                                                          |01340000
  141        |                                                                       |01350000
  142        |  /*-------------------------------------------------------------------*/|01360000
  143        |  /* This is the table containing the results when THIS program    */|01370000
  144        |  /* is the input dataset.  There is an intentional error on the   */|01380000
  145        |  /* IF count so that an error message can be produced.            */|01390000
  146        |  /*-------------------------------------------------------------------*/|01400000
  147        |  DECLARE CONTROLLED_SET(SIZE_WORD_LIST) FIXED BINARY(15)              |01410000
  148        |                              INIT( 0, 3,                              |01420000
  149        |                                    0, 1, 13, 24, 0, 2,                |01430000
  150        |                                    14,                                |01440000
  151        |                                    13, 23, 0, 0, 0, 0, 1, 0, 14,      |01450000
  152        |                                    0, 7, 0, 4, 1,                     |01460000
  153        |                                    2, 3,                              |01470000
  154        |                                    2, 4, 0, 0, 1, 0,                  |01480000
  155        |                                    2, 13, 0, 2, 0);                   |01490000
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE   6
 LINE
  157        |  /*================================================================*/|01510000
  158        |  /*= SAMPLE will perform the following tasks:                     =*/|01520000
  159        |  /*= 1) OPEN the input dataset                                   =*/|01530000
  160        |  /*= 2) READ each record and, for each record,                   =*/|01540000
  161        |  /*=    a) Extract a character string that meets the PL/I        =*/|01550000
  162        |  /*=       definition of a word.                                 =*/|01560000
  163        |  /*=    b) If the word also appears in the list of interesting   =*/|01570000
  164        |  /*=       words, record its presence by incrementing a counter. =*/|01580000
  165        |  /*= 3) Report on the number of appearances of the words that    =*/|01590000
  166        |  /*=    actually appeared in the dataset.                        =*/|01600000
  167        |  /*= 4) DISPLAY a message if the count does not match the count  =*/|01610000
  168        |  /*=    of PL/I statement keywords in this program.              =*/|01620000
  169        |  /*================================================================*/|01630000
  170        |                                                                     |01640000
  171        |  /*----------------------------------------------------------------*/|01650000
  172        |  /* Describe the action to take on selected exceptional conditions. */|01660000
  173        |  /*----------------------------------------------------------------*/|01670000
  174        |                                                                     |01680000
  175        |  /*----------------------------------------------------------------*/|01690000
  176        |  /* If the file has not been properly defined, tell them about it.  */|01700000
  177        |  /*----------------------------------------------------------------*/|01710000
  178        |  ON UNDEFINEDFILE(SOURCE)                                           |01720000
  179        |    BEGIN;                                                           |01730000
  180        |      DISPLAY ('The input data set has not been defined.');          |01740000
  181        |      STOP;                                                          |01750000
  182        |    END;                                                             |01760000
  183        |                                                                     |01770000
  184        |  /*----------------------------------------------------------------*/|01780000
  185        |  /* When the file has been processed indicate "no record read".     */|01790000
  186        |  /*----------------------------------------------------------------*/|01800000
  187        |  ON ENDFILE(SOURCE)                                                 |01810000
  188        |    BEGIN;                                                           |01820000
  189        |      RECORD_READ = FALSE;                                           |01830000
  190        |    END;                                                             |01840000
  191        |                                                                     |01850000
  192        |  /*----------------------------------------------------------------*/|01860000
  193        |  /* If any other errors occur, write a message and terminate.       */|01870000
  194        |  /*----------------------------------------------------------------*/|01880000
  195        |  ON ERROR                                                           |01890000
  196        |    BEGIN;                                                           |01900000
  197        |      ON ERROR SYSTEM;                                               |01910000
  198        |      DISPLAY ('Unspecified error occurred.  ONCODE=' || ONCODE );   |01920000
  199        |      STOP;                                                          |01930000
  200        |    END;                                                             |01940000
  201        |                                                                     |01950000
  202        |  /*----------------------------------------------------------------*/|01960000
  203        |  /* Prepare the input dataset for processing -- mark it as open.    */|01970000
  204        |  /*----------------------------------------------------------------*/|01980000
  205        |  OPEN FILE(SOURCE) INPUT;                                           |01990000
```

```
 LINE
  206 |  /*================================================================*/ 02000000
  207 |  /* Count the use of PL/I statements in each record of the         */ 02010000
  208 |  /* input data set.                                                */ 02020000
  209 |  /*================================================================*/ 02030000
  210 |                                                                      02040000
  211 |  /*----------------------------------------------------------------*/ 02050000
  212 |  /* Read the first record of the input dataset.                    */ 02060000
  213 |  /*----------------------------------------------------------------*/ 02070000
  214 |  RECORD_READ = TRUE;                                                 02080000
  215 |  READ FILE(SOURCE) INTO (RECORD);                                    02090000
  216 |                                                                      02100000
  217 |  /*----------------------------------------------------------------*/ 02110000
  218 |  /* Process the first and all succeeding records.                  */ 02120000
  219 |  /*----------------------------------------------------------------*/ 02130000
  220 |  DO WHILE (RECORD_READ);                                             02140000
  221 |                                                                      02150000
  222 |    /* Set the "last character" position to the left margin         */ 02160000
  223 |    LAST_CHAR_POSN = LEFT_MARGIN;                                     02170000
  224 |    /* Use NEXT_WORD to extract the first word from this record.    */ 02180000
  225 |    WORD = NEXT_WORD(RECORD);                                         02190000
  226 |                                                                      02200000
  227 |    /*--------------------------------------------------------------*/ 02210000
  228 |    /* Extract words from this record until no more remain.         */ 02220000
  229 |    /*--------------------------------------------------------------*/ 02230000
  230 |    DO WHILE (WORD ¬= '');                                            02240000
  231 |      /* Use LOOKUP_WORD to find its position in the table.         */ 02250000
  232 |      WORD_INDEX = LOOKUP_WORD(WORD);                                 02260000
  233 |                                                                      02270000
  234 |      /*------------------------------------------------------------*/ 02280000
  235 |      /* If the word is in the list, count it.                      */ 02290000
  236 |      /*------------------------------------------------------------*/ 02300000
  237 |      IF WORD_INDEX ¬= 0 THEN                                         02310000
  238 |        WORD_COUNT(WORD_INDEX) = WORD_COUNT(WORD_INDEX) + 1;          02320000
  239 |      ELSE;                                                           02330000
  240 |      /* Get the next word from the record.                         */ 02340000
  241 |      WORD = NEXT_WORD(RECORD);                                       02350000
  242 |    END;                                                              02360000
  243 |                                                                      02370000
  244 |    /*--------------------------------------------------------------*/ 02380000
  245 |    /* Read the next record from the input data set.                */ 02390000
  246 |    /*--------------------------------------------------------------*/ 02400000
  247 |    READ FILE(SOURCE) INTO (RECORD);                                  02410000
  248 |  END;                                                                02420000
  249 |                                                                      02430000
  250 |  /*----------------------------------------------------------------*/ 02440000
  251 |  /* Input from the data set is exhausted. CLOSE it.                */ 02450000
  252 |  /*----------------------------------------------------------------*/ 02460000
  253 |  CLOSE FILE(SOURCE);                                                 02470000
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE   8
 LINE
  255       |  /*================================================================*/|02490000
  256       |  /*= The report that details and summarizes the use of word in the =*/|02500000
  257       |  /*= WORD_TABLE is prepared in this section.                      =*/|02510000
  258       |  /*================================================================*/|02520000
  259       |                                                                      |02530000
  260       |  PUT SKIP LIST (' ********************** ');                        |02540000
  261       |  PUT SKIP LIST (' *** Word-use Report *** ');                        |02550000
  262       |  PUT SKIP LIST (' ********************** ');                        |02560000
  263       |  PUT SKIP LIST (' -count-   --word-- ');                             |02570000
  264       |                                                                      |02580000
  265       |  /*----------------------------------------------------------------*/|02590000
  266       |  /* Review the activity for each word in the list.                */|02600000
  267       |  /*----------------------------------------------------------------*/|02610000
  268       |  DO WORD_INDEX = 1 TO SIZE_WORD_LIST;                                |02620000
  269       |                                                                      |02630000
  270       |    /*--------------------------------------------------------------*/|02640000
  271       |    /* If the word was used then display the word and its use-count. */|02650000
  272       |    /*--------------------------------------------------------------*/|02660000
  273       |    IF WORD_COUNT(WORD_INDEX) > 0 THEN                                |02670000
  274       |      PUT SKIP EDIT (WORD_COUNT(WORD_INDEX),                          |02680000
  275       |                     WORD_TABLE(WORD_INDEX))                          |02690000
  276       |                     (F(6), X(6),A);                                  |02700000
  277       |    ELSE;                                                            |02710000
  278       |                                                                      |02720000
  279       |    /*--------------------------------------------------------------*/|02730000
  280       |    /* If there was a discrepancy between what was counted and what  */|02740000
  281       |    /* was expected then display a warning message and remember that */|02750000
  282       |    /* it had occurred.                                             */|02760000
  283       |    /*--------------------------------------------------------------*/|02770000
  284       |    IF WORD_COUNT(WORD_INDEX) ¬= CONTROLLED_SET(WORD_INDEX) THEN      |02780000
  285       |      DO;                                                            |02790000
  286       |        PUT SKIP EDIT ((12)'-',                                      |02800000
  287       |                       'The previous value should have been',        |02810000
  288       |                       CONTROLLED_SET(WORD_INDEX))                   |02820000
  289       |                       (A, A, F(6));                                 |02830000
  290       |        DISCREPANCY_OCCURRED = TRUE;                                 |02840000
  291       |      END;                                                           |02850000
  292       |    ELSE;                                                            |02860000
  293       |  END;                                                               |02870000
  294       |                                                                      |02880000
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE   9
 LINE
  295       |  /*================================================================*/|02890000
  296       |  /* Summarize word activity on this input dataset.                 */|02900000
  297       |  /*================================================================*/|02910000
  298       |                                                                      |02920000
  299       |  PUT SKIP(2) LIST ('There were ' || SUM(WORD_COUNT)                  |02930000
  300       |                    || ' references to ' || SIZE_WORD_LIST            |02940000
  301       |                    || ' words.');                                    |02950000
  302       |                                                                      |02960000
  303       |  /*----------------------------------------------------------------*/|02970000
  304       |  /* If a discrepency between one of the counts and the expected    */|02980000
  305       |  /* counts occured then display a warning message.                 */|02990000
  306       |  /*----------------------------------------------------------------*/|03000000
  307       |  IF DISCREPANCY_OCCURRED THEN                                        |03010000
  308       |    PUT SKIP(2) LIST ('There was a discrepancy in at least one of'    |03020000
  309       |                    || ' the word-counts.');                          |03030000
  310       |  ELSE;                                                               |03040000
```

```
 LINE
  311 |  /*=== NEXT_WORD ====================================================*/ 03050000
  312 |  /*=================================================================*/ 03060000
  313 |  /*===                                                           ===*/ 03070000
  314 |  /*===  Extract a word from the argument string that is passed.  ===*/ 03080000
  315 |  /*===  Return it as CHAR(31) VARYING.                           ===*/ 03090000
  316 |  /*===                                                           ===*/ 03100000
  317 |  /*===  Ignore PL/I comments and constants (strings surrounded by ===*/ 03110000
  318 |  /*===  single quotes (').  Comments and constants can not be     ===*/ 03120000
  319 |  /*===  continued but must be complete in the argument string.    ===*/ 03130000
  320 |  /*===                                                           ===*/ 03140000
  321 |  /*===  If no more words exist then a null character string will  ===*/ 03150000
  322 |  /*===  be returned.                                             ===*/ 03160000
  323 |  /*===                                                           ===*/ 03170000
  324 |  /*=================================================================*/ 03180000
  325 |  /*=================================================================*/ 03190000
  326 |                                                                       03200000
  327 |  NEXT_WORD: PROCEDURE(DATA_RECORD) RETURNS(CHAR(31) VARYING);         03210000
  328 |                                                                       03220000
  329 |    DECLARE DATA_RECORD        CHAR(*);                                03230000
  330 |    DECLARE DATA_WORD          CHAR(31) VARYING;                       03240000
  331 |    DECLARE NEXT_CHARACTER     CHAR(1);                                03250000
  332 |    DECLARE LENGTH_OF_STRING   FIXED BINARY(15);                       03260000
  333 |                                                                       03270000
  334 |    DECLARE NEXT_CHAR_POSN     FIXED BINARY(15);                       03280000
  335 |                                                                       03290000
  336 |    /*===============================================================*/ 03300000
  337 |    /*= LAST_CHAR_POSN remembers, from call to call, the point where=*/ 03310000
  338 |    /*= the search for additional words will start.  Management of  =*/ 03320000
  339 |    /*= its value is a key concern to this function.               =*/ 03330000
  340 |    /*=                                                             =*/ 03340000
  341 |    /*= Comments and constants in the argument string will be       =*/ 03350000
  342 |    /*= ignored.  If a character is found that is a legitimate PL/I =*/ 03360000
  343 |    /*= "first-character" then a word is assumed to follow.  It     =*/ 03370000
  344 |    /*= will be built by concatenating (suffixing) additional,      =*/ 03380000
  345 |    /*= legitimate "next-characters".                               =*/ 03390000
  346 |    /*===============================================================*/ 03400000
```



```
 LINE
  347 |    /*===============================================================*/ 03410000
  348 |    /*= Scan each character in the record.  Start at the position   =*/ 03420000
  349 |    /*= where scanning last terminated (LAST_CHAR_POSN) and         =*/ 03430000
  350 |    /*= continue until the end of a word or the end of the record   =*/ 03440000
  351 |    /*= is reached.                                                 =*/ 03450000
  352 |    /*===============================================================*/ 03460000
  353 |                                                                       03470000
  354 |    DATA_WORD = '';                                                    03480000
  355 |    DO NEXT_CHAR_POSN = LAST_CHAR_POSN TO RIGHT_MARGIN                 03490000
  356 |     WHILE (DATA_WORD = '');                                           03500000
  357 |      NEXT_CHARACTER = SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 1);         03510000
  358 |      SELECT (NEXT_CHARACTER);                                         03520000
  359 |                                                                       03530000
  360 |        WHEN ('/')                                                    03540000
  361 |                                                                       03550000
  362 |          /*-----------------------------------------------------*/    03560000
  363 |          /* If this turns out to be a comment then skip over it.  */ 03570000
  364 |          /*-----------------------------------------------------*/    03580000
  365 |          DO;                                                         03590000
  366 |            IF SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 2) = '/*' THEN    03600000
  367 |              NEXT_CHAR_POSN = NEXT_CHAR_POSN + 3                     03610000
  368 |                    + INDEX(SUBSTR(DATA_RECORD,NEXT_CHAR_POSN+2),'*/');03620000
  369 |            ELSE;                                                     03630000
  370 |          END;                                                        03640000
  371 |                                                                       03650000
  372 |        WHEN ('''')                                                   03660000
  373 |                                                                       03670000
  374 |          /*-----------------------------------------------------*/    03680000
  375 |          /* Skip over the constant.                            */    03690000
  376 |          /*-----------------------------------------------------*/    03700000
  377 |          NEXT_CHAR_POSN = NEXT_CHAR_POSN                             03710000
  378 |                    + INDEX(SUBSTR(DATA_RECORD,NEXT_CHAR_POSN+1),'''');03720000
```

```
 LINE
  379          /*============================================================*/ 03730000
  380          /*= This may be the start of a word.  Extract it if so.     =*/ 03740000
  381          /*============================================================*/ 03750000
  382                                                                           03760000
  383          OTHERWISE                                                        03770000
  384                                                                           03780000
  385            /*----------------------------------------------------------*/ 03790000
  386            /* This may be the start of a word.                         */ 03800000
  387            /*----------------------------------------------------------*/ 03810000
  388            DO;                                                            03820000
  389                                                                           03830000
  390              /*--------------------------------------------------------*/ 03840000
  391              /* If the next character is not acceptable as the first  */ 03850000
  392              /* character of a word then do nothing further -- our    */ 03860000
  393              /* enclosing DO will step to the next character for      */ 03870000
  394              /* further checking.                                     */ 03880000
  395              /*--------------------------------------------------------*/ 03890000
  396              IF INDEX(WORD_FIRST_CHARACTERS, NEXT_CHARACTER) = 0 THEN;    03900000
  397              ELSE                                                         03910000
  398                                                                           03920000
  399                /*------------------------------------------------------*/ 03930000
  400                /* This is the start of a word.  Collect the rest of it*/ 03940000
  401                /*------------------------------------------------------*/ 03950000
  402                                                                           03960000
  403                DO;                                                        03970000
  404                  DATA_WORD = NEXT_CHARACTER;                              03980000
  405                                                                           03990000
  406                  /*----------------------------------------------------*/ 04000000
  407                  /* Build up DATA_WORD by iteratively appending       */ 04010000
  408                  /* characters from the input argument string.  Do it */ 04020000
  409                  /* as long as the characters are acceptable PL/I     */ 04030000
  410                  /* "next=characters".                                */ 04040000
  411                  /*----------------------------------------------------*/ 04050000
  412                                                                           04060000
  413                  DO NEXT_CHAR_POSN = NEXT_CHAR_POSN+1 TO RIGHT_MARGIN    04070000
  414                   WHILE (INDEX(WORD_NEXT_CHARACTERS,                     04080000
  415                           SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1)) ¬= 0);  04090000
  416                    DATA_WORD = DATA_WORD                                 04100000
  417                                || SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1); 04110000
  418                  END;                                                     04120000
  419                  LAST_CHAR_POSN = NEXT_CHAR_POSN + 1;                     04130000
  420                END;                                                       04140000
  421              END;                                                         04150000
  422          END;   /* End of the SELECT (NEXT_CHARACTER) statement        */ 04160000
  423        END;     /* End of the DO that tries to find a word            */ 04170000
  424                                                                           04180000
  425        RETURN (DATA_WORD);                                                04190000
  426      END;                                                                 04200000
```

```
LINE
 427        /*=== LOOKUP_WORD ================================================*/ 04210000
 428        /*================================================================*/ 04220000
 429        /*==                                                            ==*/ 04230000
 430        /*==  Find the word in the WORD_TABLE that matches the          ==*/ 04240000
 431        /*==  argument string (CHAR(*) VARYING) and return the          ==*/ 04250000
 432        /*==  position of that word (its subscript) to the              ==*/ 04260000
 433        /*==  invoker (FIXED BINARY(15)).                               ==*/ 04270000
 434        /*==                                                            ==*/ 04280000
 435        /*==  If the word does not exist in the list a 0 will be        ==*/ 04290000
 436        /*==  returned.                                                 ==*/ 04300000
 437        /*==                                                            ==*/ 04310000
 438        /*================================================================*/ 04320000
 439        /*================================================================*/ 04330000
 440                                                                           04340000
 441        LOOKUP_WORD: PROCEDURE(DATA_WORD) RETURNS(FIXED BINARY(15));       04350000
 442                                                                           04360000
 443          DECLARE DATA_WORD   CHAR(*) VARYING;                             04370000
 444          DECLARE WORD_NUMBER FIXED BINARY(15);                            04380000
 445                                                                           04390000
 446          /*==============================================================*/ 04400000
 447          /*= A sequential search is used to locate the required word.  =*/ 04410000
 448          /*= WORD_INDEX_TABLE is used to start the search at the first =*/ 04420000
 449          /*= word in the list that has the same first character.       =*/ 04430000
 450          /*==============================================================*/ 04440000
 451          WORD_NUMBER = WORD_INDEX_TABLE   /* Subscript is on next line */  04450000
 452                      (INDEX (WORD_FIRST_CHARACTERS, SUBSTR(DATA_WORD,1,1)));04460000
 453                                                                           04470000
 454          /*--------------------------------------------------------------*/ 04480000
 455          /* Search words in the WORD_TABLE until the word is found or is */ 04490000
 456          /* determined to be not a part of the list -- its index number  */ 04500000
 457          /* (WORD_NUMBER) is zero.                                       */ 04510000
 458          /*--------------------------------------------------------------*/ 04520000
 459          DO WORD_NUMBER = WORD_NUMBER BY 1                                04530000
 460             UNTIL (  WORD_TABLE(WORD_NUMBER) = DATA_WORD                  04540000
 461                    | WORD_NUMBER = 0);                                    04550000
 462                                                                           04560000
 463            /*------------------------------------------------------------*/ 04570000
 464            /* If the word in the WORD_TABLE is alphabetically greater    */ 04580000
 465            /* than the argument word then a match cannot be found.  Set  */ 04590000
 466            /* the WORD_NUMBER to 0 to indicate a non=match situation.    */ 04600000
 467            /*------------------------------------------------------------*/ 04610000
 468            IF WORD_TABLE(WORD_NUMBER) > DATA_WORD THEN                    04620000
 469              WORD_NUMBER = 0;                                             04630000
 470            ELSE;                                                          04640000
 471          END;                                                             04650000
 472                                                                           04660000
 473          RETURN (WORD_NUMBER);                                            04670000
 474        END;                                                               04680000
 475        %;                                                                 04690000
 476        %/*============================================================*/; 04700000
 477        %/*============================================================*/; 04710000
```

```
LINE
 478        %/*=                                                          =*/; 04720000
 479        %/*= Submit a list of words that are to be included in the    =*/; 04730000
 480        %/*= WORD_TABLE.  They must be in alphabetic order.  The list =*/; 04740000
 481        %/*= must be within parentheses and cannot contain any blanks =*/; 04750000
 482        %/*=                                                          =*/; 04760000
 483        %/*============================================================*/; 04770000
 484        %/*============================================================*/; 04780000
 485        %ADD_TO_LIST: PROC(WORD_LIST) RETURNS(CHAR);                      04790000
 486          DCL WORD_LIST       CHAR;                                       04800000
 487          DCL EXTRACTED_WORD  CHAR;                                       04810000
 488          DCL THIS_INDEX_CHAR CHAR;                                       04820000
 489          DCL THIS_INDEX      FIXED;                                      04830000
 490          DCL COMMA           FIXED;                                      04840000
 491          DCL (INDEX, LENGTH, SUBSTR) BUILTIN;                            04850000
 492                                                                          04860000
 493          /* Remove the leading and trailing apostrophes               */ 04870000
 494          WORD_LIST = SUBSTR(WORD_LIST,2,LENGTH(WORD_LIST)-2);            04880000
 495                                                                          04890000
 496          /*------------------------------------------------------------*/ 04900000
 497          /* Extract each word from the argument data string.  Put each */ 04910000
 498          /* word onto the end of a growing string that will eventually */ 04920000
 499          /* be used in the INITIAL clause of the declaration of        */ 04930000
 500          /* the WORD_TABLE.                                            */ 04940000
 501          /*------------------------------------------------------------*/ 04950000
 502        PARSE_LOOP:                                                       04960000
 503          /* Keep track of the size of the word list.                  */ 04970000
 504          SIZE_WORD_LIST = SIZE_WORD_LIST + 1;                            04980000
 505          THIS_INDEX = INDEX('ABCDEFGHIJKLMNOPQRSTUVWXYZ',                04990000
 506                             SUBSTR(WORD_LIST,1,1));                      05000000
 507                                                                          05010000
 508          /*------------------------------------------------------------*/ 05020000
 509          /* Is this is the first time that this initial character has  */ 05030000
 510          /* been encountered?  If so, the table of indices into        */ 05040000
 511          /* WORD_TABLE must be updated.                                */ 05050000
 512          /*------------------------------------------------------------*/ 05060000
 513          IF THIS_INDEX ¬= LAST_INDEX THEN                                05070000
 514            DO;                                                           05080000
 515                                                                          05090000
 516              /*----------------------------------------------------------*/ 05100000
 517              /* Update the "initial character index" for all characters  */ 05110000
 518              /* between the last one and this one.                       */ 05120000
 519              /*----------------------------------------------------------*/ 05130000
 520                                                                          05140000
 521              DO LAST_INDEX = LAST_INDEX+1 TO THIS_INDEX;                 05150000
 522                /*--------------------------------------------------------*/ 05160000
 523                /* If the last character processed and this character are */ 05170000
 524                /* are not alphabetically adjacent then a zero (no word   */ 05180000
 525                /* having such a first character is acceptable) must be   */ 05190000
 526                /* appended to the list of indices.  The list will be     */ 05200000
 527                /* used later in the INITIAL clause of the declaration    */ 05210000
 528                /* of WORD_TABLE_INDEX.                                   */ 05220000
```

```
 LINE
  529          /*-------------------------------------------------------*/ 05230000
  530          IF LAST_INDEX < THIS_INDEX THEN                            05240000
  531            FIRST_WORD_INDICES = FIRST_WORD_INDICES || ', 0';        05250000
  532          ELSE                                                       05260000
  533          FIRST_WORD_INDICES = FIRST_WORD_INDICES||','||SIZE_WORD_LIST; 05270000
  534        END;                                                         05280000
  535        LAST_INDEX = LAST_INDEX - 1;                                 05290000
  536      END;                                                           05300000
  537    ELSE;                                                            05310000
  538                                                                     05320000
  539    COMMA = INDEX(WORD_LIST,',');                                    05330000
  540                                                                     05340000
  541    /*-----------------------------------------------------------*/ 05350000
  542    /* Is there a comma after this word?                         */ 05360000
  543    /*-----------------------------------------------------------*/ 05370000
  544    IF COMMA = 0 THEN                                                05380000
  545      DO;                                                            05390000
  546                                                                     05400000
  547        /*-------------------------------------------------------*/ 05410000
  548        /* Since this word is not followed by a comma it is the  */ 05420000
  549        /* last one in the list.                                 */ 05430000
  550        /*-------------------------------------------------------*/ 05440000
  551        BIG_LIST = BIG_LIST || '''' || WORD_LIST || ''', ';          05450000
  552                                                                     05460000
  553        /*-------------------------------------------------------*/ 05470000
  554        /* Keep track of the longest word in the list.           */ 05480000
  555        /*-------------------------------------------------------*/ 05490000
  556        IF LENGTH(WORD_LIST) > MAX_WORD_LENGTH THEN                  05500000
  557          MAX_WORD_LENGTH = LENGTH(WORD_LIST);                       05510000
  558        ELSE;                                                        05520000
  559        RETURN('');                                                  05530000
  560      END;                                                           05540000
  561    ELSE                                                             05550000
  562      DO;                                                            05560000
  563                                                                     05570000
  564        /*-------------------------------------------------------*/ 05580000
  565        /* Extract the next word and remove it from the input.   */ 05590000
  566        /*-------------------------------------------------------*/ 05600000
  567        EXTRACTED_WORD = SUBSTR(WORD_LIST,1,COMMA-1);                05610000
  568        BIG_LIST = BIG_LIST || ''''                                  05620000
  569                            || EXTRACTED_WORD || ''', ';             05630000
  570                                                                     05640000
  571        /*-------------------------------------------------------*/ 05650000
  572        /* Keep track of the longest word in the list.           */ 05660000
  573        /*-------------------------------------------------------*/ 05670000
  574        IF LENGTH(EXTRACTED_WORD) > MAX_WORD_LENGTH THEN             05680000
  575          MAX_WORD_LENGTH = LENGTH(EXTRACTED_WORD);                  05690000
  576        ELSE;                                                        05700000
  577        /* Remove this word and the comma from the input string. */ 05710000
  578        WORD_LIST = SUBSTR(WORD_LIST,COMMA+1);                       05720000
  579      END;                                                           05730000
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation         */ PAGE  16
 LINE
  580        |    GO TO PARSE_LOOP;                                                                    |05740000
  581        |  %END;                                                                                  |05750000
  582        |  %;                                                                                     |05760000
  583        |  %/*=================================================================*/;                  |05770000
  584        |  %/*=================================================================*/;                  |05780000
  585        |  %/*=                                                               =*/;                |05790000
  586        |  %/*= All words contained in the search list have been submitted.   =*/;                |05800000
  587        |  %/*= Create the DECLAREs for the WORD_TABLE, WORD_COUNT vector     =*/;                |05810000
  588        |  %/*= and the WORD_INDEX_TABLE.                                     =*/;                |05820000
  589        |  %/*=                                                               =*/;                |05830000
  590        |  %/*=================================================================*/;                  |05840000
  591        |  %/*=================================================================*/;                  |05850000
  592        |  %END_OF_LIST: PROC RETURNS(CHAR);                                                      |05860000
  593        |    DCL TABLE_DCL CHAR;                                                                  |05870000
  594        |                                                                                         |05880000
  595        |    /*-------------------------------------------------------------*/                    |05890000
  596        |    /* Create the DECLARE for the WORD_TABLE                      */                    |05900000
  597        |    /*-------------------------------------------------------------*/                    |05910000
  598        |    TABLE_DCL = 'DECLARE '                                                               |05920000
  599        |                || 'WORD_TABLE(' || (SIZE_WORD_LIST+1) || ') '                           |05930000
  600        |                || 'CHAR(' || MAX_WORD_LENGTH || ')'                                     |05940000
  601        |                || 'INIT(' || BIG_LIST                                                   |05950000
  602        |                           || 'HIGH(' || MAX_WORD_LENGTH || '));' ;                      |05960000
  603        |                                                                                         |05970000
  604        |    /*-------------------------------------------------------------*/                    |05980000
  605        |    /* Append the DECLARE for the WORD_COUNT array.               */                    |05990000
  606        |    /*-------------------------------------------------------------*/                    |06000000
  607        |    TABLE_DCL = TABLE_DCL                                                                |06010000
  608        |             || 'DECLARE WORD_COUNT(' || SIZE_WORD_LIST || ') '                          |06020000
  609        |             || 'FIXED BINARY(15) INIT((' || SIZE_WORD_LIST || ')0);' ;                  |06030000
  610        |                                                                                         |06040000
  611        |    /*-------------------------------------------------------------*/                    |06050000
  612        |    /* Append the DECLARE for the WORD_INDEX_TABLE array.         */                    |06060000
  613        |    /*-------------------------------------------------------------*/                    |06070000
  614        |    TABLE_DCL = TABLE_DCL                                                                |06080000
  615        |             || 'DECLARE WORD_INDEX_TABLE(26) FIXED BINARY(15) INIT(' ;                  |06090000
  616        |    TABLE_DCL = TABLE_DCL || SUBSTR(FIRST_WORD_INDICES,5);                               |06100000
  617        |                                                                                         |06110000
  618        |    /*-------------------------------------------------------------*/                    |06120000
  619        |    /* If the last word started with a Z then the initial values  */                    |06130000
  620        |    /* for the index table is complete.  If not then some zeroes  */                    |06140000
  621        |    /* have to be added to account for all 26 array items.        */                    |06150000
  622        |    /*-------------------------------------------------------------*/                    |06160000
  623        |    IF SIZE_WORD_LIST = 26 THEN                                                          |06170000
  624        |      TABLE_DCL = TABLE_DCL || ')' ;                                                     |06180000
  625        |    ELSE                                                                                 |06190000
  626        |      TABLE_DCL = TABLE_DCL || ',(' || 26-LAST_INDEX || ')0);';                          |06200000
  627        |    NOTE ('The WORD_TABLE was successfully declared.',0);                                |06205000
  628        |    RETURN(TABLE_DCL);                                                                   |06210000
  629        |  %END;                                                                                  |06220000
  630        |END;                                                                                     |06230000

5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation         */ PAGE  17
PREPROCESSOR DIAGNOSTIC MESSAGES
   1   2   3
ERROR ID L   LINE    MESSAGE DESCRIPTION


PREPROCESSOR INFORMATORY MESSAGES

IEL2250I I   140     The WORD_TABLE was successfully declared.

END OF PREPROCESSOR DIAGNOSTIC MESSAGES
```

Diagnostic messages generated by the preprocessor. All messages generated by the compiler (including the preprocessor) are documented in the *PL/I VSE Compile-Time Messages and Codes* book.

1 ERROR ID identifies the message originating from the compiler (IEL) and gives the message number.

2 L indicates the severity level of the message.

3 LINE lists the number of the line in which the error occurred.

```
                SOURCE LISTING                                                  4
   STMT LEV NT                                                                  R

               |/* PL/I Sample Program: Used to verify product installation         */|00040000

               |/*=== SAMPLE ===========================================================*/|00060000
               |/*=====================================================================*/|00070000
               |/***                                                                 ==*/|00080000
               |/***  This is the PL/I sample program that is intended to be      ==*/|00090000
               |/***  used to verify the product's complete installation.        ==*/|00100000
               |/***  It is expected to execute and to provide some output.      ==*/|00110000
               |/***  Although "results" are created by the program it is only   ==*/|00120000
               |/***  to verify that representative I/O services are operable --  ==*/|00130000
               |/***  the results are verified (internally) by the program.      ==*/|00140000
               |/***                                                             ==*/|00150000
               |/***  The program is intended to read a data file and count      ==*/|00160000
               |/***  the number of occurrences of each PL/I statement type.     ==*/|00170000
               |/***  The results are displayed at the end of execution.         ==*/|00180000
               |/***  If any count does not match the value that is expected     ==*/|00190000
               |/***  a warning message is displayed.                            ==*/|00200000
               |/***                                                             ==*/|00210000
               |/***  When the program is executed this source program file will  ==*/|00220000
               |/***  be used as the input file.  The filename or DDNAME is      ==*/|00230000
               |/***  SOURCE.                                                    ==*/|00240000
               |/***                                                             ==*/|00250000
               |/***  NOTE: Compilation of this program should cause preprocessor ==*/|00260000
               |/***        message:                                             ==*/|00270000
               |/***                                                             ==*/|00280000
               |/***   IEL2250I I 140 The WORD_TABLE was successfully declared.  ==*/|00281000
               |/***                                                             ==*/|00282000
               |/***        Two compiler messages will be produced as well:      ==*/|00283000
               |/***                                                             ==*/|00284000
               |/***   IEL0533I I     NO 'DECLARE' STATEMENT(S) FOR 'INDEX'.     ==*/|00285000
               |/***   IEL0871I I  62 RESULT OF BUILTIN FUNCTION 'SUM' WILL BE   ==*/|00286000
               |/***                  EVALUATED USING FIXED POINT ARITHMETIC     ==*/|00287000
               |/***                  OPERATIONS.                               ==*/|00288000
               |/***                                                             ==*/|00289000
               |/*=====================================================================*/|00290000
               |/*=====================================================================*/|00300000

     1      0  |SAMPLE: PROCEDURE OPTIONS(MAIN) REORDER;                             |00320000
```

Source listing. This is the output from the preprocessor and the input to the compiler. All the preprocessor statements have been executed and all preprocessor comments have been deleted.

4 Numbers in this column of the listing indicate the maximum depth of replacement of preprocessor statements.

```
                  /*================================================================*/ 00640000
                  /* Non-Preprocessor data variables are declared here.  Only the   */ 00650000
                  /* variables that are used in the main block (or in more than one */ 00660000
                  /* of the contained blocks) are defined here.                     */ 00670000
                  /*================================================================*/ 00680000
                                                                                       00690000
                  /*----------------------------------------------------------------*/ 00700000
                  /* Declare the source program input file and its accoutrements.   */ 00710000
                  /*----------------------------------------------------------------*/ 00720000
   2   1  0       DECLARE SOURCE         FILE RECORD ENVIRONMENT(F RECSIZE(80));       00730000
   3   1  0       DECLARE RECORD         CHARACTER(80);                                00740000
   4   1  0       DECLARE RECORD_READ    BIT(1) INIT(FALSE);                           00750000
   5   1  0       DECLARE LAST_CHAR_POSN FIXED BINARY(15);                             00760000
   6   1  0       DECLARE DISCREPANCY_OCCURRED  BIT(1) INIT(FALSE);                    00770000
                                                                                       00780000
                  /*----------------------------------------------------------------*/ 00790000
                  /* Declare the left- and right-margins of the input dataset.      */ 00800000
                  /*----------------------------------------------------------------*/ 00810000
   7   1  0       DECLARE LEFT_MARGIN    FIXED BINARY(15) INIT('2');                   00820000
   8   1  0       DECLARE RIGHT_MARGIN   FIXED BINARY(15) INIT('72');                  00830000
                                                                                       00840000
                  /*----------------------------------------------------------------*/ 00850000
                  /* Declare '1'B as TRUE and '0'B as FALSE.                        */ 00860000
                  /*----------------------------------------------------------------*/ 00870000
   9   1  0       DECLARE TRUE           BIT(1) INIT('1'B);                            00880000
  10   1  0       DECLARE FALSE          BIT(1) INIT('0'B);                            00890000
                                                                                       00900000
                  /*----------------------------------------------------------------*/ 00910000
                  /* Declare which characters are acceptable as the first character */ 00920000
                  /* of a word -- then declare acceptable succeeding characters.    */ 00930000
                  /*----------------------------------------------------------------*/ 00940000
  11   1  0       DECLARE WORD_FIRST_CHARACTERS CHAR(29) STATIC                        00950000
                             INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ@#$');                    00960000
  12   1  0       DECLARE WORD_NEXT_CHARACTERS  CHAR(30) STATIC                        00970000
                             INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ_@#$');                   00980000
                                                                                       00990000
                  /*----------------------------------------------------------------*/ 01000000
                  /* Declare a place to hold words extracted from program text.     */ 01010000
                  /*----------------------------------------------------------------*/ 01020000
  13   1  0       DECLARE WORD  CHAR(31) VARYING;                                      01030000
  14   1  0       DECLARE WORD_INDEX     FIXED BINARY(15);                             01040000
                                                                                       01050000
                  /*----------------------------------------------------------------*/ 01060000
                  /* Declare the use of SYSPRINT and all of the builtin functions.  */ 01070000
                  /*----------------------------------------------------------------*/ 01080000
  15   1  0       DECLARE SYSPRINT       FILE STREAM OUTPUT PRINT ENV(MEDIUM(SYSLST)),  01090000
                         PLIXOPT CHAR(100) VAR STATIC EXT INIT('MSGFILE(SYSPRINT)');   01095000
  16   1  0       DECLARE (HIGH, SUBSTR, SUM, UNSPEC, VERIFY) BUILTIN;                 01100000
  17   1  0       DECLARE ONCODE         BUILTIN;                                      01110000
```

```
STMT LEV NT                                                                  R
                  /*================================================================*/ 01120000
                  /* PL/I statement keywords are collected using the ADD_TO_LIST    */ 01130000
                  /* macro.  They are put into a table, WORD_TABLE, by the          */ 01140000
                  /* END_OF_LIST macro.  That macro also creates an index,          */ 01150000
                  /* WORD_TABLE_INDEX, into the WORD_TABLE.                         */ 01160000
                  /*                                                                */ 01170000
                  /* Finally, a table, WORD_COUNT, is created that has a counter    */ 01180000
                  /* that corresponds to each word.  Whenever that word is          */ 01190000
                  /* encountered in the input stream the appropriate WORD_COUNT     */ 01200000
                  /* element is incremented.                                        */ 01210000
                  /*                                                                */ 01220000
                  /* Notice that there are no semicolons in the macro statements.   */ 01230000
                  /*================================================================*/ 01240000
                                                                                    01260000  1
                                                                                    01270000  1
                                                                                    01280000  1
                                                                                    01290000  1
                                                                                    01300000  1
                                                                                    01310000  1
                                                                                    01320000  1
                                                                                    01330000  1
   18   1  0     DECLARE WORD_TABLE(      37) CHAR(       9)INIT('ALLOCATE', 'BEGIN', 01340000  1
                'CALL', 'CLOSE', 'DCL', 'DECLARE', 'DEFAULT', 'DISPLAY', 'DO', 'ELSE', 01340000  1
                'END', 'ENTRY', 'FREE', 'GENERIC', 'GET', 'GO', 'GOTO', 'IF', 'LEAVE', 01340000  1
                'LIST', 'LOCATE', 'ON', 'OPEN', 'PROC', 'PROCEDURE', 'READ', 'RETURN', 01340000  1
                'REVERT', 'REWRITE', 'SELECT', 'SIGNAL', 'STOP', 'THEN', 'WAIT', 'WHEN' 01340000  1
   19   1  0     , 'WRITE', HIGH(       9));DECLARE WORD_COUNT(      36) FIXED BINARY(15 01340000  1
   20   1  0     ) INIT((      36)0);DECLARE WORD_INDEX_TABLE(26) FIXED BINARY(15) INIT( 01340000  1
                     1,       2,       3,       5,      10,      13,      14, 0,      18 01340000  1
                , 0, 0,      19, 0, 0,      22,      24, 0,      26,      30,      33, 01340000  1
                0, 0,      34,(       3)0);                                          01340000  1
                                                                                    01350000
                  /*----------------------------------------------------------------*/ 01360000
                  /* This is the table containing the results when THIS program     */ 01370000
                  /* is the input dataset.  There is an intentional error on the    */ 01380000
                  /* IF count so that an error message can be produced.             */ 01390000
                  /*----------------------------------------------------------------*/ 01400000
   21   1  0     DECLARE CONTROLLED_SET(      36) FIXED BINARY(15)                  01410000  1
                                          INIT( 0, 3,                               01420000
                                                0, 1, 13, 24, 0, 2,                 01430000
                                                14,                                 01440000
                                                13, 23, 0, 0, 0, 0, 1, 0, 14,       01450000
                                                0, 7, 0, 4, 1,                      01460000
                                                2, 3,                               01470000
                                                2, 4, 0, 0, 1, 0,                   01480000
                                                2, 13, 0, 2, 0);                    01490000
```

```
                  /*================================================================*/ 01510000
                  /*= SAMPLE will perform the following tasks:                    =*/ 01520000
                  /*= 1) OPEN the input dataset                                   =*/ 01530000
                  /*= 2) READ each record and, for each record,                   =*/ 01540000
                  /*=    a) Extract a character string that meets the PL/I        =*/ 01550000
                  /*=       definition of a word.                                 =*/ 01560000
                  /*=    b) If the word also appears in the list of interesting   =*/ 01570000
                  /*=       words, record its presence by incrementing a counter. =*/ 01580000
                  /*= 3) Report on the number of appearances of the words that    =*/ 01590000
                  /*=    actually appeared in the dataset.                        =*/ 01600000
                  /*= 4) DISPLAY a message if the count does not match the count  =*/ 01610000
                  /*=    of PL/I statement keywords in this program.              =*/ 01620000
                  /*================================================================*/ 01630000
                                                                                      01640000
                  /*----------------------------------------------------------------*/ 01650000
                  /* Describe the action to take on selected exceptional conditions. */ 01660000
                  /*----------------------------------------------------------------*/ 01670000
                                                                                      01680000
                  /*----------------------------------------------------------------*/ 01690000
                  /* If the file has not been properly defined, tell them about it. */ 01700000
                  /*----------------------------------------------------------------*/ 01710000
22  1  0          ON UNDEFINEDFILE(SOURCE)                                            01720000
                    BEGIN;                                                            01730000
23  2  0            DISPLAY ('The input data set has not been defined.');             01740000
24  2  0            STOP;                                                             01750000
25  2  0           END;                                                               01760000
                                                                                      01770000
                  /*----------------------------------------------------------------*/ 01780000
                  /* When the file has been processed indicate "no record read".    */ 01790000
                  /*----------------------------------------------------------------*/ 01800000
26  1  0          ON ENDFILE(SOURCE)                                                  01810000
                    BEGIN;                                                            01820000
27  2  0            RECORD_READ = FALSE;                                              01830000
28  2  0           END;                                                               01840000
                                                                                      01850000
                  /*----------------------------------------------------------------*/ 01860000
                  /* If any other errors occur, write a message and terminate.      */ 01870000
                  /*----------------------------------------------------------------*/ 01880000
29  1  0          ON ERROR                                                            01890000
                    BEGIN;                                                            01900000
30  2  0            ON ERROR SYSTEM;                                                  01910000
31  2  0            DISPLAY ('Unspecified error occurred.  ONCODE=' || ONCODE );      01920000
32  2  0            STOP;                                                             01930000
33  2  0           END;                                                               01940000
                                                                                      01950000
                  /*----------------------------------------------------------------*/ 01960000
                  /* Prepare the input dataset for processing -- mark it as open.   */ 01970000
                  /*----------------------------------------------------------------*/ 01980000
34  1  0          OPEN FILE(SOURCE) INPUT;                                            01990000
```

```
   STMT LEV NT                                                                    R
                  /*====================================================================*/ 02000000
                  /* Count the use of PL/I statements in each record of the          */ 02010000
                  /* input data set.                                                 */ 02020000
                  /*====================================================================*/ 02030000
                                                                                      02040000
                  /*--------------------------------------------------------------------*/ 02050000
                  /* Read the first record of the input dataset.                     */ 02060000
                  /*--------------------------------------------------------------------*/ 02070000
     35   1  0    RECORD_READ = TRUE;                                                 02080000
     36   1  0    READ FILE(SOURCE) INTO (RECORD);                                    02090000
                                                                                      02100000
                  /*--------------------------------------------------------------------*/ 02110000
                  /* Process the first and all succeeding records.                   */ 02120000
                  /*--------------------------------------------------------------------*/ 02130000
     37   1  0    DO WHILE (RECORD_READ);                                             02140000
                                                                                      02150000
                    /* Set the "last character" position to the left margin          */ 02160000
     38   1  1      LAST_CHAR_POSN = LEFT_MARGIN;                                     02170000
                    /* Use NEXT_WORD to extract the first word from this record.     */ 02180000
     39   1  1      WORD = NEXT_WORD(RECORD);                                         02190000
                                                                                      02200000
                    /*------------------------------------------------------------------*/ 02210000
                    /* Extract words from this record until no more remain.          */ 02220000
                    /*------------------------------------------------------------------*/ 02230000
     40   1  1      DO WHILE (WORD ¬= '');                                            02240000
                      /* Use LOOKUP_WORD to find its position in the table.          */ 02250000
     41   1  2        WORD_INDEX = LOOKUP_WORD(WORD);                                 02260000
                                                                                      02270000
                      /*----------------------------------------------------------------*/ 02280000
                      /* If the word is in the list, count it.                       */ 02290000
                      /*----------------------------------------------------------------*/ 02300000
     42   1  2        IF WORD_INDEX ¬= 0 THEN                                         02310000
                        WORD_COUNT(WORD_INDEX) = WORD_COUNT(WORD_INDEX) + 1;          02320000
     43   1  2        ELSE;                                                           02330000
                        /* Get the next word from the record.                        */ 02340000
     44   1  2        WORD = NEXT_WORD(RECORD);                                       02350000
     45   1  2      END;                                                              02360000
                                                                                      02370000
                    /*------------------------------------------------------------------*/ 02380000
                    /* Read the next record from the input data set.                 */ 02390000
                    /*------------------------------------------------------------------*/ 02400000
     46   1  1      READ FILE(SOURCE) INTO (RECORD);                                02410000
     47   1  1    END;                                                              02420000
                                                                                      02430000
                  /*--------------------------------------------------------------------*/ 02440000
                  /* Input from the data set is exhausted. CLOSE it.                 */ 02450000
                  /*--------------------------------------------------------------------*/ 02460000
     48   1  0    CLOSE FILE(SOURCE);                                                 02470000
```

```
STMT LEV NT                                                                                R

                 /*==============================================================*/
                 /*= The report that details and summarizes the use of word in the =*/
                 /*= WORD_TABLE is prepared in this section.                     =*/
                 /*==============================================================*/

  49   1  0      PUT SKIP LIST (' *********************** ');
  50   1  0      PUT SKIP LIST (' *** Word-use Report *** ');
  51   1  0      PUT SKIP LIST (' *********************** ');
  52   1  0      PUT SKIP LIST (' -count-   --word-- ');

                 /*--------------------------------------------------------------*/
                 /* Review the activity for each word in the list.               */
                 /*--------------------------------------------------------------*/
  53   1  0      DO WORD_INDEX = 1 TO       36;                                         1

                   /*------------------------------------------------------------*/
                   /* If the word was used then display the word and its use-count. */
                   /*------------------------------------------------------------*/
  54   1  1        IF WORD_COUNT(WORD_INDEX) > 0 THEN
                     PUT SKIP EDIT (WORD_COUNT(WORD_INDEX),
                                    WORD_TABLE(WORD_INDEX))
                                    (F(6), X(6),A);
  55   1  1        ELSE;

                   /*------------------------------------------------------------*/
                   /* If there was a discrepancy between what was counted and what  */
                   /* was expected then display a warning message and remember that */
                   /* it had occurred.                                           */
                   /*------------------------------------------------------------*/
  56   1  1        IF WORD_COUNT(WORD_INDEX) ¬= CONTROLLED_SET(WORD_INDEX) THEN
                     DO;
  57   1  2            PUT SKIP EDIT ((12)'-',
                                      'The previous value should have been',
                                      CONTROLLED_SET(WORD_INDEX))
                                      (A, A, F(6));
  58   1  2          DISCREPANCY_OCCURRED = TRUE;
  59   1  2        END;
  60   1  1      ELSE;
  61   1  1    END;
```



```
STMT LEV NT                                                                                R
                 /*==============================================================*/
                 /* Summarize word activity on this input dataset.               */
                 /*==============================================================*/

  62   1  0      PUT SKIP(2) LIST ('There were ' || SUM(WORD_COUNT)
                                  || ' references to ' ||       36                    1
                                  || ' words.');

                 /*--------------------------------------------------------------*/
                 /* If a discrepency between one of the counts and the expected   */
                 /* counts occured then display a warning message.               */
                 /*--------------------------------------------------------------*/
  63   1  0      IF DISCREPANCY_OCCURRED THEN
                   PUT SKIP(2) LIST ('There was a discrepancy in at least one of'
                                  || ' the word-counts.');
  64   1  0      ELSE;
```

```
   STMT LEV NT                                                                                                  R
                  /*=== NEXT_WORD ========================================================*/ 03050000
                  /*=====================================================================*/ 03060000
                  /*==                                                                ==*/ 03070000
                  /*==  Extract a word from the argument string that is passed.       ==*/ 03080000
                  /*==  Return it as CHAR(31) VARYING.                                ==*/ 03090000
                  /*==                                                                ==*/ 03100000
                  /*==  Ignore PL/I comments and constants (strings surrounded by     ==*/ 03110000
                  /*==  single quotes (').  Comments and constants can not be         ==*/ 03120000
                  /*==  continued but must be complete in the argument string.        ==*/ 03130000
                  /*==                                                                ==*/ 03140000
                  /*==  If no more words exist then a null character string will      ==*/ 03150000
                  /*==  be returned.                                                  ==*/ 03160000
                  /*==                                                                ==*/ 03170000
                  /*=====================================================================*/ 03180000
                  /*=====================================================================*/ 03190000
                                                                                           03200000
     65   1  0    NEXT_WORD: PROCEDURE(DATA_RECORD) RETURNS(CHAR(31) VARYING);            03210000
                                                                                           03220000
     66   2  0      DECLARE DATA_RECORD        CHAR(*);                                   03230000
     67   2  0      DECLARE DATA_WORD          CHAR(31) VARYING;                          03240000
     68   2  0      DECLARE NEXT_CHARACTER     CHAR(1);                                   03250000
     69   2  0      DECLARE LENGTH_OF_STRING   FIXED BINARY(15);                          03260000
                                                                                           03270000
     70   2  0      DECLARE NEXT_CHAR_POSN     FIXED BINARY(15);                          03280000
                                                                                           03290000
                    /*=================================================================*/ 03300000
                    /*= LAST_CHAR_POSN remembers, from call to call, the point where=*/ 03310000
                    /*= the search for additional words will start.  Management of  =*/ 03320000
                    /*= its value is a key concern to this function.                =*/ 03330000
                    /*=                                                             =*/ 03340000
                    /*= Comments and constants in the argument string will be       =*/ 03350000
                    /*= ignored.  If a character is found that is a legitimate PL/I =*/ 03360000
                    /*= "first-character" then a word is assumed to follow.  It     =*/ 03370000
                    /*= will be built by concatenating (suffixing) additional,      =*/ 03380000
                    /*= legitimate "next-characters".                               =*/ 03390000
                    /*=================================================================*/ 03400000
```



```
   STMT LEV NT                                                                                                  R
                    /*=================================================================*/ 03410000
                    /*= Scan each character in the record.  Start at the position   =*/ 03420000
                    /*= where scanning last terminated (LAST_CHAR_POSN) and         =*/ 03430000
                    /*= continue until the end of a word or the end of the record   =*/ 03440000
                    /*= is reached.                                                 =*/ 03450000
                    /*=================================================================*/ 03460000
                                                                                           03470000
     71   2  0      DATA_WORD = '';                                                       03480000
     72   2  0      DO NEXT_CHAR_POSN = LAST_CHAR_POSN TO RIGHT_MARGIN                    03490000
                     WHILE (DATA_WORD = '');                                             03500000
     73   2  1       NEXT_CHARACTER = SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 1);           03510000
     74   2  1       SELECT (NEXT_CHARACTER);                                           03520000
                                                                                           03530000
     75   2  2         WHEN ('/')                                                       03540000
                                                                                           03550000
                        /*-------------------------------------------------------*/ 03560000
                        /* If this turns out to be a comment then skip over it.  */ 03570000
                        /*-------------------------------------------------------*/ 03580000
                        DO;                                                            03590000
     76   2  3             IF SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 2) = '/*' THEN      03600000
                              NEXT_CHAR_POSN = NEXT_CHAR_POSN + 3                      03610000
                                   + INDEX(SUBSTR(DATA_RECORD,NEXT_CHAR_POSN+2),'*/'); 03620000
     77   2  3             ELSE;                                                       03630000
     78   2  3             END;                                                        03640000
                                                                                           03650000
     79   2  2         WHEN ('''')                                                      03660000
                                                                                           03670000
                        /*-------------------------------------------------------*/ 03680000
                        /* Skip over the constant.                               */ 03690000
                        /*-------------------------------------------------------*/ 03700000
                        NEXT_CHAR_POSN = NEXT_CHAR_POSN                                03710000
                                 + INDEX(SUBSTR(DATA_RECORD,NEXT_CHAR_POSN+1),''''); 03720000
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE  27
   STMT LEV NT                                                                                 R
                        /*=======================================================*/
                        /*= This may be the start of a word.  Extract it if so.     =*/
                        /*=======================================================*/

     80   2  2          OTHERWISE

                          /*-----------------------------------------------------*/
                          /* This may be the start of a word.                    */
                          /*-----------------------------------------------------*/
                          DO;

                            /*---------------------------------------------------*/
                            /* If the next character is not acceptable as the first  */
                            /* character of a word then do nothing further -- our    */
                            /* enclosing DO will step to the next character for      */
                            /* further checking.                                      */
                            /*---------------------------------------------------*/
     81   2  3              IF INDEX(WORD_FIRST_CHARACTERS, NEXT_CHARACTER) = 0 THEN;
     82   2  3              ELSE

                              /*-------------------------------------------------*/
                              /* This is the start of a word.  Collect the rest of it*/
                              /*-------------------------------------------------*/

                              DO;
     83   2  4                  DATA_WORD = NEXT_CHARACTER;

                                /*-----------------------------------------------*/
                                /* Build up DATA_WORD by iteratively appending       */
                                /* characters from the input argument string.  Do it */
                                /* as long as the characters are acceptable PL/I     */
                                /* "next=characters".                                 */
                                /*-----------------------------------------------*/

     84   2  4                  DO NEXT_CHAR_POSN = NEXT_CHAR_POSN+1 TO RIGHT_MARGIN
                                 WHILE (INDEX(WORD_NEXT_CHARACTERS,
                                         SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1)) ¬= 0);
     85   2  5                    DATA_WORD = DATA_WORD
                                             || SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1);
     86   2  5                  END;
     87   2  4                  LAST_CHAR_POSN = NEXT_CHAR_POSN + 1;
     88   2  4                END;
     89   2  3            END;
     90   2  2        END;  /* End of the SELECT (NEXT_CHARACTER) statement         */
     91   2  1      END;    /* End of the DO that tries to find a word               */

     92   2  0      RETURN (DATA_WORD);
     93   2  0    END;
```

```
STMT LEV NT                                                                    R
                  /*=== LOOKUP_WORD ================================================*/ 04210000
                  /*================================================================*/ 04220000
                  /*==                                                            ==*/ 04230000
                  /*==  Find the word in the WORD_TABLE that matches the          ==*/ 04240000
                  /*==  argument string (CHAR(*) VARYING) and return the          ==*/ 04250000
                  /*==  position of that word (its subscript) to the              ==*/ 04260000
                  /*==  invoker (FIXED BINARY(15)).                               ==*/ 04270000
                  /*==                                                            ==*/ 04280000
                  /*==  If the word does not exist in the list a 0 will be        ==*/ 04290000
                  /*==  returned.                                                 ==*/ 04300000
                  /*==                                                            ==*/ 04310000
                  /*================================================================*/ 04320000
                  /*================================================================*/ 04330000
                                                                                     04340000
   94   1  0      LOOKUP_WORD: PROCEDURE(DATA_WORD) RETURNS(FIXED BINARY(15));       04350000
                                                                                     04360000
   95   2  0        DECLARE DATA_WORD   CHAR(*) VARYING;                            04370000
   96   2  0        DECLARE WORD_NUMBER FIXED BINARY(15);                           04380000
                                                                                     04390000
                    /*================================================================*/ 04400000
                    /*= A sequential search is used to locate the required word.    =*/ 04410000
                    /*= WORD_INDEX_TABLE is used to start the search at the first   =*/ 04420000
                    /*= word in the list that has the same first character.         =*/ 04430000
                    /*================================================================*/ 04440000
   97   2  0        WORD_NUMBER = WORD_INDEX_TABLE   /* Subscript is on next line */   04450000
                                (INDEX (WORD_FIRST_CHARACTERS, SUBSTR(DATA_WORD,1,1)));04460000
                                                                                     04470000
                    /*----------------------------------------------------------------*/ 04480000
                    /* Search words in the WORD_TABLE until the word is found or is  */ 04490000
                    /* determined to be not a part of the list -- its index number   */ 04500000
                    /* (WORD_NUMBER) is zero.                                        */ 04510000
                    /*----------------------------------------------------------------*/ 04520000
   98   2  0        DO WORD_NUMBER = WORD_NUMBER BY 1                               04530000
                       UNTIL (  WORD_TABLE(WORD_NUMBER) = DATA_WORD                  04540000
                              | WORD_NUMBER = 0);                                    04550000
                                                                                     04560000
                      /*----------------------------------------------------------------*/ 04570000
                      /* If the word in the WORD_TABLE is alphabetically greater     */ 04580000
                      /* than the argument word then a match cannot be found.  Set   */ 04590000
                      /* the WORD_NUMBER to 0 to indicate a non=match situation.     */ 04600000
                      /*----------------------------------------------------------------*/ 04610000
   99   2  1          IF WORD_TABLE(WORD_NUMBER) > DATA_WORD THEN                     04620000
                        WORD_NUMBER = 0;                                             04630000
  100   2  1          ELSE;                                                          04640000
  101   2  1        END;                                                             04650000
                                                                                     04660000
  102   2  0        RETURN (WORD_NUMBER);                                            04670000
  103   2  0      END;                                                               04680000
  104   1  0    END;                                                                 06230000
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE  29
                                                    ATTRIBUTE AND CROSS-REFERENCE TABLE (SHORT)
DCL NO.    IDENTIFIER                    ATTRIBUTES AND REFERENCES

21         CONTROLLED_SET                (36) AUTOMATIC ALIGNED INITIAL BINARY FIXED (15,0)
                                         1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,56,
                                         57
66         DATA_RECORD                   /* PARAMETER */ UNALIGNED CHARACTER(*)
                                         73,76,76,79,84,85
95         DATA_WORD                     /* PARAMETER */ UNALIGNED CHARACTER(*) VARYING
                                         97,98,99
67         DATA_WORD                     AUTOMATIC UNALIGNED CHARACTER (31) VARYING
                                         71,72,83,85,85,92
6          DISCREPANCY_OCCURRED          AUTOMATIC UNALIGNED INITIAL BIT (1)
                                         1,58,63
10         FALSE                         AUTOMATIC UNALIGNED INITIAL BIT (1)
                                         1,1,1,27
16         HIGH                          BUILTIN
                                         18
********   INDEX                         BUILTIN
                                         76,79,81,84,97
5          LAST_CHAR_POSN                AUTOMATIC ALIGNED BINARY FIXED (15,0)
                                         38,72,87
7          LEFT_MARGIN                   AUTOMATIC ALIGNED INITIAL BINARY FIXED (15,0)
                                         1,38
94         LOOKUP_WORD                   ENTRY RETURNS(BINARY FIXED (15,0))
                                         41
70         NEXT_CHAR_POSN                AUTOMATIC ALIGNED BINARY FIXED (15,0)
                                         72,72,73,76,76,76,76,79,79,79,84,84,84,84,85,87
68         NEXT_CHARACTER                AUTOMATIC UNALIGNED CHARACTER (1)
                                         73,74,81,83
65         NEXT_WORD                     ENTRY RETURNS(CHARACTER (31) VARYING )
                                         39,44
```

Attribute and cross-reference table

1. Number of the statement in the source listing in which the identifier is explicitly declared.
2. Asterisks indicate an undeclared identifier: all of its attributes are implied or supplied by default.
3. All identifiers used in the program are listed in ascending order according to their binary value.
4. Declared and default attributes are listed. This list also includes descriptive comments.
5. Cross-references: these are the numbers of all other statements in which the identifier appears.

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE  30
DCL NO.    IDENTIFIER                     ATTRIBUTES AND REFERENCES
17         ONCODE                         BUILTIN
                                          31
3          RECORD                         AUTOMATIC UNALIGNED CHARACTER (80)
                                          36,39,44,46
4          RECORD_READ                    AUTOMATIC UNALIGNED INITIAL BIT (1)
                                          1,35,37
                                          27
8          RIGHT_MARGIN                   AUTOMATIC ALIGNED INITIAL BINARY FIXED (15,0)
                                          1,72,84
2          SOURCE                         EXTERNAL FILE RECORD ENVIRONMENT(F RECSIZE(80))
                                          22,26,34,36,46,48
16         SUBSTR                         BUILTIN
                                          73,76,76,79,84,85,97
16         SUM                            BUILTIN
                                          62
15         SYSPRINT                       EXTERNAL FILE STREAM PRINT OUTPUT ENVIRONMENT(MEDIUM(SYSLST))
                                          49,50,51,52,54,57,62,63
9          TRUE                           AUTOMATIC UNALIGNED INITIAL BIT (1)
                                          1,35,58
13         WORD                           AUTOMATIC UNALIGNED CHARACTER (31) VARYING
                                          39,40,41,44
19         WORD_COUNT                     (36) AUTOMATIC ALIGNED INITIAL BINARY FIXED (15,0)
                                          1,42,42,54,54,56,62
11         WORD_FIRST_CHARACTERS          STATIC UNALIGNED INITIAL CHARACTER (29)
                                          81,97
14         WORD_INDEX                     AUTOMATIC ALIGNED BINARY FIXED (15,0)
                                          41,42,42,42,53,53,54,54,54,56,56,57
20         WORD_INDEX_TABLE               (26) AUTOMATIC ALIGNED INITIAL BINARY FIXED (15,0)
                                          1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,97
12         WORD_NEXT_CHARACTERS           STATIC UNALIGNED INITIAL CHARACTER (30)
                                          84
96         WORD_NUMBER                    AUTOMATIC ALIGNED BINARY FIXED (15,0)
                                          97,98,98,98,98,98,99,99,102

5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE  31
DCL NO.    IDENTIFIER                     ATTRIBUTES AND REFERENCES
18         WORD_TABLE                     (37) AUTOMATIC UNALIGNED INITIAL CHARACTER (9)
                                          1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                                          54,98,99

5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE  32
 1          2                        AGGREGATE LENGTH TABLE                    3
DCL NO.    IDENTIFIER                      LVL     DIMS        OFFSET       ELEMENT        TOTAL
                                                                             LENGTH        LENGTH

21         CONTROLLED_SET                            1                          2           72
19         WORD_COUNT                                1                          2           72
20         WORD_INDEX_TABLE                          1                          2           52
18         WORD_TABLE                                1                          9          333
                                                       SUM OF CONSTANT LENGTHS              529  4
```

Aggregate length table

1. Number of the statement in which the aggregate is declared, or, for a controlled aggregate, the number of the associated ALLOCATE statement.
2. The elements of the aggregate as declared.
3. Length of each element of the aggregate.
4. Sum of the lengths of aggregates whose lengths are constant.

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE  33
                    STORAGE REQUIREMENTS
 5                                    6
BLOCK, SECTION OR STATEMENT          TYPE                  LENGTH 7 (HEX)     DSA SIZE 8 (HEX)
*SAMPLE1                             PROGRAM CSECT           4420    1144
*SAMPLE2                             STATIC CSECT            2888     B48
SAMPLE                               PROCEDURE BLOCK         2298    8FA         1160     488
BLOCK 2       STMT 22                ON UNIT                  170     AA          208      D0
BLOCK 3       STMT 26                ON UNIT                  162     A2          216      D8
BLOCK 4       STMT 29                ON UNIT                  304    130          288     120
NEXT_WORD                            PROCEDURE BLOCK         1032    408          368     170
LOOKUP_WORD                          PROCEDURE BLOCK          448    1C0          256     100
```

Storage requirements. This table gives the main storage requirements for the program. These quantities do not include the main storage required by the library subroutines that will be included by the linkage editor or loaded dynamically during execution.

5 Name of the block, section, or number of the statement in the program.

6 Description of the block, section, or statement.

7 Length in bytes of the storage areas in both decimal and hexadecimal notation.

8 Length in bytes of the dynamic storage area (DSA) in both decimal and hexadecimal notation.

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE  34
                    EXTERNAL SYMBOL DICTIONARY
          1              2          3          4          5
        SYMBOL         TYPE        ID       ADDR     LENGTH

       CEESTART         SD      0001     000000     000080
       *SAMPLE1         SD      0002     000000     001144
       *SAMPLE2         SD      0003     000000     000B48
       CEEMAIN          WX      0004     000000
       CEEMAIN          SD      0005     000000     000010
       IBMRINP1         ER      0006     000000
       CEEFMAIN         WX      0007     000000
       CEEBETBL         ER      0008     000000
       CEEROOTA         ER      0009     000000
       CEESG010         ER      000A     000000
       IELCGOG          SD      000B     000000     0000AE
       IELCGOH          SD      000C     000000     0000A0
       IELCGOC          SD      000D     000000     00007C
       IELCGMY          SD      000E     000000     0000A4
       IELCGCY          SD      000F     000000     00007E
       IBMSSIOA         ER      0010     000000
       IBMSASCA         ER      0011     000000
       IBMSCEDB         ER      0012     000000
       IBMSCHFD         ER      0013     000000
       IBMSCHXH         WX      0014     000000
       IBMSCWDH         ER      0015     000000
       IBMSEOCA         ER      0016     000000
       IBMSJDSA         ER      0017     000000
       IBMSOCLA         ER      0018     000000
       IBMSOCLC         WX      0019     000000
       IBMSRIOA         ER      001A     000000
       IBMSSEOA         ER      001B     000000
       IBMSSIOE         WX      001C     000000
       IBMSSIOT         WX      001D     000000
       IBMSSLOA         ER      001E     000000
       IBMSSPLA         ER      001F     000000
       IBMSSXCA         ER      0020     000000
       IBMSSXCB         WX      0021     000000
       IBMSSIST         WX      0022     000000
       CEEUOPT          SD      0023     000000     0004D8
       PLIXOPT          SD      0024     000000     000066
       PLIXOPT*         SD      0025     000000     00002C
       PLIXOPT+         LD               000004
       PLIXOPT-         LD               000020
       PLIXOPT-         ER      0026     000000
       SAMPLE           LD               000008
       SAMPLE           ER      0027     000000
       SAMPLE*          SD      0028     000000     000020
       SAMPLE+          LD               000000
       SOURCE           SD      0029     000000     000068
       SOURCE           PR      002A     000000     000004
       SOURCE*          SD      002B     000000     000020

5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation        */ PAGE  35
       SOURCE+          LD               000000
       SYSPINT          SD      002C     000000     00006E
       SYSPINT*         SD      002D     000000     000020
       SYSPINT+         LD               000000
```

External symbol dictionary

1 List of all the external symbols that make up the object module.

2 Type of external symbol, as follows:
- CM Common area
- ER External reference
- LD Label definition
- PR Pseudo-register
- SD Section definition
- WX Weak external reference

Full definitions of these terms are given in "External symbol dictionary" in the main text.

3 All entries, except LD type entries, are identified by a hexadecimal number.

4 Address (in hexadecimal) of LD type entries.

5 Length in bytes (in hexadecimal) of SD, CM, and PR type entries.

STATIC INTERNAL STORAGE MAP

1	2	3
000000	E0000B40	PROGRAM ADCON
000004	00000008	PROGRAM ADCON
000008	000000FC	PROGRAM ADCON
00000C	0000034C	PROGRAM ADCON
000010	000008FC	PROGRAM ADCON
000014	0000096E	PROGRAM ADCON
000018	000009A8	PROGRAM ADCON
00001C	00000A1A	PROGRAM ADCON
000020	00000A24	PROGRAM ADCON
000024	00000A4C	PROGRAM ADCON
000028	00000AD8	PROGRAM ADCON
00002C	00000B88	PROGRAM ADCON
000030	00000C1A	PROGRAM ADCON
000034	00000C40	PROGRAM ADCON
000038	00000F90	PROGRAM ADCON
00003C	0000101A	PROGRAM ADCON
000040	00001024	PROGRAM ADCON
000044	00001024	PROGRAM ADCON
000048	00001024	PROGRAM ADCON
00004C	00001024	PROGRAM ADCON
000050	00001024	PROGRAM ADCON
000054	00001024	PROGRAM ADCON
000058	00001024	PROGRAM ADCON
00005C	00000000	A..IELCGOG
000060	00000000	A..IELCGOH
000064	00000000	A..IELCGOC
000068	00000000	A..IELCGMY
00006C	00000000	A..IELCGCY
000070	00000000	A..IBMSASCA
000074	00000000	A..IBMSCEDB
000078	00000000	A..IBMSCHFD
00007C	00000000	A..IBMSCHXH
000080	00000000	A..IBMSCWDH
000084	00000000	A..IBMSEOCA
000088	00000000	A..IBMSJDSA
00008C	00000000	A..IBMSOCLA
000090	00000000	A..IBMSOCLC
000094	00000000	A..IBMSRIOA
000098	00000000	A..IBMSSEOA
00009C	00000000	A..IBMSSIOE
0000A0	00000000	A..IBMSSIOT
0000A4	00000000	A..IBMSSLOA
0000A8	00000000	A..IBMSSPLA
0000AC	00000000	A..IBMSSXCA
0000B0	00000000	A..IBMSSXCB
0000B4	00000000	A..STATIC
0000B8	B4000A00	DED..NEXT_WORD
0000BC	2000	DED
0000BE	00000F80	DED..WORD_COUNT
0000C2	500000060080	FED
0000C8	60000006	FED
0000CC	58000009	FED
0000D0	5800000C	FED
0000D4	58000023	FED
0000D8	2800	DED..WORD
0000DA	2401	DED..FALSE
0000DC	0002	CONSTANT
0000DE	0048	CONSTANT
0000E0	0000	CONSTANT
0000E2	0001	CONSTANT
0000E4	0003	CONSTANT
0000E6	0005	CONSTANT
0000E8	000A	CONSTANT
0000EA	000D	CONSTANT
0000EC	000E	CONSTANT
0000EE	0012	CONSTANT
0000F0	0013	CONSTANT
0000F2	0016	CONSTANT
0000F4	0018	CONSTANT
0000F6	001A	CONSTANT
0000F8	001E	CONSTANT
0000FA	0021	CONSTANT
0000FC	0022	CONSTANT
0000FE	0017	CONSTANT
000100	0007	CONSTANT
000102	0004	CONSTANT
000104	0024	CONSTANT
000106	0009	CONSTANT
000108	4040402020202020	CONSTANT
	202020202120	
000116	001C	CONSTANT
000118	001D	CONSTANT
00011A		
000120	00000000000001BC	LOCATOR..CONTROLLED_SET
000128	00000000000001CC	LOCATOR..WORD_INDEX_TABLE
000130	00000000000001DC	LOCATOR..WORD_TABLE
000138	00000000001F8000	LOCATOR..WORD
000140	0000053D001E0000	LOCATOR..WORD_NEXT_CHARACTERS
000148	00000520001D0000	LOCATOR..WORD_FIRST_CHARACTERS
000150	0000000000010000	LOCATOR..FALSE
000158	0000000000500000	LOCATOR..RECORD
000160	0000000000000000	LOCATOR..DATA_RECORD
000168	0000000000008000	LOCATOR..DATA_WORD
000170	0080000091102000	CONSTANT
000178	0000000000000050	RECORD DESCRIPTOR
000180	000003E000190000	LOCATOR
000188	000003F900190000	LOCATOR
000190	0000041200140000	LOCATOR
000198	0000000000340000	LOCATOR
0001A0	00000000003B0000	LOCATOR
0001A8	000004B600280000	LOCATOR
0001B0	00000000002D0000	LOCATOR

Static internal storage map.
This is a map of the static control section for the program. This control section is the third standard entry in the external symbol dictionary.

1 Six-digit offset (in hexadecimal)

2 Text (in hexadecimal)

3 Comment indicating type of item to which the text refers. A comment appears only against the first line of the text for an item.

```
0001B8  91E091E0            CONSTANT
0001BC  0000000200000002    DESCRIPTOR
        0000002400000001
0001CC  0000000200000002    DESCRIPTOR
        0000001A00000001
0001DC  0000000900000009    DESCRIPTOR
        0000002500000001
        00090000
0001F0  00000001            CONSTANT
0001F4  0010000000601800    CONSTANT
        00000000
000200  00000002            CONSTANT
000204  00000003            CONSTANT
000208  0000001F            CONSTANT
00020C  00000000            A..PLIXOPT
000210  00000000            A..DCLCB
000214  00000000            A..DCLCB
000218  000001F0            A..CONSTANT
00021C  00000000            A..DCLCB
000220  000001F4            A..CONSTANT
000224  00000000            OMITTED ARGUMENT
000228  00000000            OMITTED ARGUMENT
00022C  80000000            OMITTED ARGUMENT
000230  00000000            A..DCLCB
000234  00000170            A..CONSTANT
000238  00000000            A..RD
00023C  00000000            OMITTED ARGUMENT
000240  00000000            OMITTED ARGUMENT
000244  80000000            OMITTED ARGUMENT
000248  00000000            A..LOCATOR
00024C  80000000            A..TEMP
000250  00000000            A..LOCATOR
000254  80000000            A..WORD_INDEX
000258  000001F0            A..CONSTANT
00025C  00000000            A..DCLCB
000260  80000000            OMITTED ARGUMENT
000264  00000000            A..DCLCB
000268  00000000            A..TEMP
00026C  800001F0            A..CONSTANT
000270  00000000            A..DCLCB
000274  00000000            A..TEMP
000278  80000200            A..CONSTANT
00027C  00000000            A..LOCATOR
000280  000000E2            A..CONSTANT
000284  000000BE            A..DED..WORD_COUNT
000288  00000000            A..TEMP
00028C  800000E0            A..CONSTANT
000290  800001A8            A..CONSTANT
000294  0D800000            CONSTANT
000298  80000000            A..TEMP
00029C  C1D3D3D6C3C1E3C5    CONSTANT
        40
0002A5  C2C5C7C9D5404040    CONSTANT
        40
0002AE  C3C1D3D340404040    CONSTANT
        40
0002B7  C3D3D6E2C5404040    CONSTANT
        40
0002C0  C4C3D34040404040    CONSTANT
        40
0002C9  C4C5C3D3C1D9C540    CONSTANT
        40
0002D2  C4C5C6C1E4D3E340    CONSTANT
        40
0002DB  C4C9E2D7D3C1E840    CONSTANT
        40
0002E4  C4D6404040404040    CONSTANT
        40
0002ED  C5D3E2C540404040    CONSTANT
        40
0002F6  C5D5C44040404040    CONSTANT
        40
0002FF  C5D5E3D9E8404040    CONSTANT
        40
000308  C6D9C5C540404040    CONSTANT
        40
000311  C7C5D5C5D9C9C340    CONSTANT
        40
00031A  C7C5E34040404040    CONSTANT
        40
000323  C7D6404040404040    CONSTANT
        40
00032C  C7D6E3D640404040    CONSTANT
        40
000335  C9C6404040404040    CONSTANT
        40
00033E  D3C5C1E5C5404040    CONSTANT
        40
000347  D3C9E2E340404040    CONSTANT
        40
000350  D3D6C3C1E3C54040    CONSTANT
        40
000359  D6D5404040404040    CONSTANT
        40
000362  D6D7C5D540404040    CONSTANT
        40
00036B  D7D9D6C340404040    CONSTANT
        40
000374  D7D9D6C3C5C4E4D9    CONSTANT
        C5
00037D  D9C5C1C440404040    CONSTANT
        40
000386  D9C5E3E4D9D54040    CONSTANT
        40
```

```
00038F  D9C5E5C5D9E34040      CONSTANT
        40
000398  D9C5E6D9C9E3C540      CONSTANT
        40
0003A1  E2C5D3C5C3E34040      CONSTANT
        40
0003AA  E2C9C7D5C1D34040      CONSTANT
        40
0003B3  E2E3D6D740404040      CONSTANT
        40
0003BC  E3C8C5D540404040      CONSTANT
        40
0003C5  E6C1C9E340404040      CONSTANT
        40
0003CE  E6C8C5D540404040      CONSTANT
        40
0003D7  E6D9C9E3C5404040      CONSTANT
        40
0003E0  405C5C5C5C5C5C5C      CONSTANT
        5C5C5C5C5C5C5C5C
        5C5C5C5C5C5C5C5C
        40
0003F9  405C5C5C40E69699      CONSTANT
        8460A4A28540D985
        979699A3405C5C5C
        40
000412  40608396A495A360      CONSTANT
        4040406060A69699
        84606040
000426  6060606060606060      CONSTANT
        60606060
000432  E3888540979985A5      CONSTANT
        8996A4A240A58193
        A48540A28896A493
        84408881A5854082
        858595
000455  E38885998540A685      CONSTANT
        998540
000460  4099858685998595      CONSTANT
        8385A240A39640
00046F  404040F3F6            CONSTANT
000474  40A6969984A24B        CONSTANT
00047B  E38885998540A681      CONSTANT
        A24081408489A283
        998597819583A840
        89954081A3409385
        81A2A34096958540
        968640A3888540A6
        969984608396A495
        A3A24B
0004B6  E3888540899597A4      CONSTANT
        A3408481A38140A2
        85A3408881A24095
        96A3408285859540
        8485868995858 44B
0004DE  4040202020202021      CONSTANT
        20
0004E7  E495A29785838986      CONSTANT
        8985844085999996
        9940968383A49999
        85844B4040D6D5C3
        D6C4C57E
00050B  615C                  CONSTANT
00050D  5C61                  CONSTANT
00050F  7D                    CONSTANT
000510  0C16000000000A4C      STATIC ONCB
000518  0C96000000000000      STATIC ONCB
000520  C1C2C3C4C5C6C7C8      INITIAL VALUE..WORD_FIRST_CHAR
        C9D1D2D3D4D5D6D7
        D8D9E2E3E4E5E6E7
        E8E97C7B5B
00053D  C1C2C3C4C5C6C7C8      INITIAL VALUE..WORD_NEXT_CHARA
        C9D1D2D3D4D5D6D7
        D8D9E2E3E4E5E6E7
        E8E96D7C7B5B
000560  00000000              SYMBOL TABLE ELEMENT
000564  00000000              CONSTANT
000568  00000000              SYMBOL TABLE ELEMENT
00056C  0000060C              SYMBOL TABLE ELEMENT
000570  0000062C              SYMBOL TABLE ELEMENT
000574  00000650              SYMBOL TABLE ELEMENT
000578  0000066C              SYMBOL TABLE ELEMENT
00057C  80000000              SYMBOL TABLE ELEMENT
000580  00000688              SYMBOL TABLE ELEMENT
000584  000006A4              SYMBOL TABLE ELEMENT
000588  000006BC              SYMBOL TABLE ELEMENT
00058C  000006E4              SYMBOL TABLE ELEMENT
000590  0000070C              SYMBOL TABLE ELEMENT
000594  00000724              SYMBOL TABLE ELEMENT
000598  0000073C              SYMBOL TABLE ELEMENT
00059C  0000075C              SYMBOL TABLE ELEMENT
0005A0  0000077C              SYMBOL TABLE ELEMENT
0005A4  000007A4              SYMBOL TABLE ELEMENT
0005A8  000007C4              SYMBOL TABLE ELEMENT
0005AC  000007E4              SYMBOL TABLE ELEMENT
0005B0  000008D8              SYMBOL TABLE ELEMENT
0005B4  000008F4              SYMBOL TABLE ELEMENT
0005B8  00000000              SYMBOL TABLE ELEMENT
0005BC  00000000              SYMBOL TABLE ELEMENT
0005C0  00000000              CONSTANT
0005C4  00000560              SYMBOL TABLE ELEMENT
0005C8  00000000              CONSTANT
0005CC  0000056C              SYMBOL TABLE ELEMENT
0005D0  00000000              CONSTANT
```

```
0005D4  0000056C            SYMBOL TABLE ELEMENT                000000F000000000
0005D8  00000000            CONSTANT                            0004E3D9E4C50000
0005DC  0000056C            SYMBOL TABLE ELEMENT          00073C 85000001000000BE      SYMBOL TABLE..RIGHT_MARGIN
0005E0  000007FC            SYMBOL TABLE ELEMENT                0000011A00000000
0005E4  0000081C            SYMBOL TABLE ELEMENT                000CD9C9C7C8E36D
0005E8  00000840            SYMBOL TABLE ELEMENT                D4C1D9C7C9D50000
0005EC  00000860            SYMBOL TABLE ELEMENT          00075C 85000001000000BE      SYMBOL TABLE..LEFT_MARGIN
0005F0  0000087C            SYMBOL TABLE ELEMENT                0000011C00000000
0005F4  00000000            CONSTANT                            000BD3C5C6E36DD4
0005F8  0000056C            SYMBOL TABLE ELEMENT                C1D9C7C9D5000000
0005FC  0000089C            SYMBOL TABLE ELEMENT          00077C 81000001000000DA      SYMBOL TABLE..DISCREPANCY_OCCU
000600  000008BC            SYMBOL TABLE ELEMENT                000000F800000000
000604  00000000            CONSTANT                            0014C4C9E2C3D9C5
000608  0000056C            SYMBOL TABLE ELEMENT                D7C1D5C3E86DD6C3
00060C  81000101000000BE    SYMBOL TABLE..CONTROLLED_SET        C3E4D9D9C5C40000
        000000C000000000                                  0007A4 85000001000000BE      SYMBOL TABLE..LAST_CHAR_POSN
        000EC3D6D5E3D9D6                                         0000011E00000000
        D3D3C5C46DE2C5E3                                         000ED3C1E2E36DC3
00062C  81000101000000BE    SYMBOL TABLE..WORD_INDEX_TABLE      C8C1D96DD7D6E2D5
        000000C800000000                                  0007C4 81000001000000DA      SYMBOL TABLE..RECORD_READ
        0010E6D6D9C46DC9                                         0000010000000000
        D5C4C5E76DE3C1C2                                         000BD9C5C3D6D9C4
        D3C50000                                                 6DD9C5C1C4000000
000650  81000101000000BE    SYMBOL TABLE..WORD_COUNT      0007E4 81000001000000BC      SYMBOL TABLE..RECORD
        000000D000000000                                         0000010800000000
        000AE6D6D9C46DC3                                         0006D9C5C3D6D9C4
        D6E4D5E3                                          0007FC 85000002000000BE      SYMBOL TABLE..NEXT_CHAR_POSN
00066C  81000101000000BC    SYMBOL TABLE..WORD_TABLE            000000D000000000
        000000D800000000                                         000ED5C5E7E36DC3
        000AE6D6D9C46DE3                                         C8C1D96DD7D6E2D5
        C1C2D3C5                                          00081C 85000002000000BE      SYMBOL TABLE..LENGTH_OF_STRING
000688  85000001000000BE    SYMBOL TABLE..WORD_INDEX            000000D200000000
        0000011800000000                                         0010D3C5D5C7E3C8
        000AE6D6D9C46DC9                                         6DD6C66DE2E3D9C9
        D5C4C5E7                                                 D5C70000
0006A4  81000001000000D8    SYMBOL TABLE..WORD            000840 81000002000000BC      SYMBOL TABLE..NEXT_CHARACTER
        000000E000000000                                         000000B800000000
        0004E6D6D9C40000                                         000ED5C5E7E36DC3
0006BC  01000000000000BC    SYMBOL TABLE..WORD_NEXT_CHARAC      C8C1D9C1C3E3C5D9
        0000014000000000                                  000860 81000002000000D8      SYMBOL TABLE..DATA_WORD
        0014E6D6D9C46DD5                                         000000C000000000
        C5E7E36DC3C8C1D9                                         0009C4C1E3C16DE6
        C1C3E3C5D9E20000                                         D6D9C400
0006E4  01000000000000BC    SYMBOL TABLE..WORD_FIRST_CHARA00087C A1000002000000BC      SYMBOL TABLE..DATA_RECORD
        0000014800000000                                         0000010000000000
        0015E6D6D9C46DC6                                         000BC4C1E3C16DD9
        C9D9E2E36DC3C8C1                                         C5C3D6D9C4000000
        D9C1C3E3C5D9E200                                  00089C 85000002000000BE      SYMBOL TABLE..WORD_NUMBER
00070C  81000001000000DA    SYMBOL TABLE..FALSE                 000000C000000000
        000000E800000000                                         000BE6D6D9C46DD5
        0005C6C1D3E2C500                                         E4D4C2C5D9000000
000724  81000001000000DA    SYMBOL TABLE..TRUE            0008BC A1000002000000D8      SYMBOL TABLE..DATA_WORD
```

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation         */ PAGE  40
        000000D000000000
        0009C4C1E3C16DE6                                        000000  0011D4E2C7C6C9D3      CSECT FOR EXTERNAL VARIABLE
        D6D9C400                                                        C54DE2E8E2D7D9C9
0008D8  0D020001000000B8      SYMBOL TABLE..NEXT_WORD                   D5E35D0000000000
        00000B8800000000                                                0000000000000000
        0009D5C5E7E36DE6                                                0000000000000000
        D6D9C400                                                        0000000000000000
0008F4  0D020001000000B8      SYMBOL TABLE..LOOKUP_WORD                 0000000000000000
        00000F9000000000                                                0000000000000000
        000BD3D6D6D2E4D7                                                0000000000000000
        6DE6D6D9C4000000                                                0000000000000000
                                                                        0000000000000000
                                                                        0000000000000000
                                                                        000000000000

                          STATIC EXTERNAL CSECTS                000000  2800                  SYMTAB DED..PLIXOPT
                                                                000002  0000
                                                                000004  1900000000000000      SYMBOL TABLE..PLIXOPT
                                                                        0000002400000000
000000  FFFFFFFC41201000      DCLCB                                     0007D7D3C9E7D6D7
        02D70F0000000020                                                E3000000
        000000140008E2E8                                        000020  80000004              SYMBOL TABLE ELEMENT
        E2D7D9C9D5E30000                                        000024  0000000000648000      LOCATOR..PLIXOPT
000020  0000000000000000      ENVB
        0100000200000064                                        000000  1D02000000000018      SYMBOL TABLE..SAMPLE
        0000006400000064                                                0000000000000000
        0000006400000064                                                0006E2C1D4D7D3C5
        0000006400000064                                        000018  B4000A00              SYMTAB DED..SAMPLE
        0000006400000064
        0000006400000064                                        000000  1D00000100000018      SYMBOL TABLE..SOURCE
        0000006800000064                                                0000000000000000
        00000064                                                        0006E2D6E4D9C3C5
000064  00000000              ENVB CONSTANT                     000018  B800                  SYMTAB DED..SOURCE
000068  E2E8E2D3E2E3          ENVB CONSTANT
                                                                000000  1D0000010000001C      SYMBOL TABLE..SYSPRINT
                                                                        0000000000000000
000000  0000000002000000      DCLCB                                     0008E2E8E2D7D9C9
        010010000000001C                                                D5E30000
        000000140006E2D6                                        00001C  B800                  SYMTAB DED..SYSPRINT
        E4D9C3C5
00001C  0080000000008000      ENVB
        0000000200000060
        0000006400000060
        0000006000000060
        0000006000000060
        0000006000000060
        0000006000000060
        0000006000000060
        00000060
000060  00000000              ENVB CONSTANT
000064  00000050              ENVB CONSTANT

5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation         */ PAGE  41
                              VARIABLE STORAGE MAP
IDENTIFIER                    LEVEL        OFFSET       (HEX)   CLASS     BLOCK

CONTROLLED_SET                    1           296         128   AUTO      SAMPLE
WORD_INDEX_TABLE                  1           368         170   AUTO      SAMPLE
WORD_COUNT                        1           420         1A4   AUTO      SAMPLE
WORD_TABLE                        1           526         20E   AUTO      SAMPLE
WORD_INDEX                        1           280         118   AUTO      SAMPLE
WORD                              1           492         1EC   AUTO      SAMPLE
WORD_NEXT_CHARACTERS              1          1341         53D   STATIC    SAMPLE
WORD_FIRST_CHARACTERS             1          1312         520   STATIC    SAMPLE
FALSE                             1           288         120   AUTO      SAMPLE
TRUE                              1           289         121   AUTO      SAMPLE
RIGHT_MARGIN                      1           282         11A   AUTO      SAMPLE
LEFT_MARGIN                       1           284         11C   AUTO      SAMPLE
DISCREPANCY_OCCURRED              1           290         122   AUTO      SAMPLE
LAST_CHAR_POSN                    1           286         11E   AUTO      SAMPLE
RECORD_READ                       1           291         123   AUTO      SAMPLE
RECORD                            1           859         35B   AUTO      SAMPLE
NEXT_CHAR_POSN                    2           208          D0   AUTO      NEXT_WORD
LENGTH_OF_STRING                  2           210          D2   AUTO      NEXT_WORD
NEXT_CHARACTER                    2           212          D4   AUTO      NEXT_WORD
DATA_WORD                         2           216          D8   AUTO      NEXT_WORD
WORD_NUMBER                       2           192          C0   AUTO      LOOKUP_WORD
```

TABLES OF OFFSETS AND STATEMENT NUMBERS

WITHIN PROCEDURE SAMPLE

OFFSET (HEX)	0	348	358	368	370	37E	388	39E	3AE	3BA	410	432	462	48A	492
STATEMENT NO.	1	22	26	29	34	35	36	37	38	39	40	41	42	43	44
OFFSET (HEX)	4E8	4F0	506	50E	51C	55C	59C	5DC	61C	62E	63A	642	6DA	6E2	6FA
STATEMENT NO.	45	46	47	48	49	50	51	52	53	54	53	54	55	56	57
OFFSET (HEX)	78A	794	79C	7A4	7A8	7B4	7CC	7CC	86E	8D2	8DA				
STATEMENT NO.	58	59	60	61	54	61	53	62	63	64	104				

WITHIN ON UNIT BLOCK 2

OFFSET (HEX)	0	76	84	92
STATEMENT NO.	22	23	24	25

WITHIN ON UNIT BLOCK 3

OFFSET (HEX)	0	80	8A
STATEMENT NO.	26	27	28

WITHIN ON UNIT BLOCK 4

OFFSET (HEX)	0	90	98	10A	118
STATEMENT NO.	29	30	31	32	33

WITHIN PROCEDURE NEXT_WORD

OFFSET (HEX)	0	BC	C8	100	118	122	132	1B4	1BC	1C4	238	240	27E	286	298
STATEMENT NO.	65	71	72	73	74	75	76	77	78	79	80	81	82	83	84
OFFSET (HEX)	312	36E	372	376	38C	3A0	3A4	3A8	3A8	3AC	3C4	3FC			
STATEMENT NO.	85	86	84	86	87	88	89	90	90	91	92	93			

WITHIN PROCEDURE LOOKUP_WORD

OFFSET (HEX)	0	98	DA	E2	132	13A	13E	146	148	15E	176	182	184	18C	190
STATEMENT NO.	94	97	98	99	100	101	98	101	98	101	98	101	98	101	102
OFFSET (HEX)	1B4														
STATEMENT NO.	103														

```
      1
* STATEMENT NUMBER  40                          2
000418  44 00 C 1AC              3  EX    0,HOOK..STMT
00041C                       CL.48  EQU   *
00041C  48 90 3 0E0                 LH    9,224(0,3)
000420  48 70 D 1EC                 LH    7,WORD
000424  41 80 3 29C                 LA    8,668(0,3)
000428  41 60 D 1EE                 LA    6,WORD+2
00042C  58 F0 3 06C                 L     15,A..IELCGCY
000430  05 EF                       BALR  14,15
000432  47 80 2 1AC                 BE    CL.15
000436  44 00 C 1C8                 EX    0,HOOK..DO


* STATEMENT NUMBER  41
00043A  44 00 C 1AC                 EX    0,HOOK..STMT
00043E  41 70 D 0E0                 LA    7,224(0,13)
000442  50 70 3 250                 ST    7,592(0,3)
000446  41 70 D 118                 LA    7,WORD_INDEX
00044A  50 70 3 254                 ST    7,596(0,3)
00044E  96 80 3 254                 OI    596(3),X'80'
000452  18 5D                       LR    5,13
000454  41 10 3 250                 LA    1,592(0,3)
000458  58 F0 3 038                 L     15,A..LOOKUP_WORD
00045C  44 00 C 1C0              3  EX    0,HOOK..PRE-CALL
000460  47 01 4 00A                 NOP   HOOK..INFO
000464  05 EF                       BALR  14,15
000466  44 00 C 1C4              3  EX    0,HOOK..POST-CALL


* STATEMENT NUMBER  42
00046A  44 00 C 1AC                 EX    0,HOOK..STMT
00046E  48 90 D 118                 LH    9,WORD_INDEX
000472  49 90 3 0E0                 CH    9,224(0,3)
000476  47 80 2 146                 BE    CL.16
00047A  44 00 C 1CC                 EX    0,HOOK..IF-TRUE
00047E  89 90 0 001                 SLL   9,1
000482  48 69 D 1A2                 LH    6,VO..WORD_COUNT(9
                                          )
000486  4A 60 3 0E2                 AH    6,226(0,3)
00048A  40 69 D 1A2                 STH   6,VO..WORD_COUNT(9
                                          )
00048E  47 F0 2 14E                 B     CL.17


* STATEMENT NUMBER  43
000492                       CL.16  EQU   *
000492  44 00 C 1AC                 EX    0,HOOK..STMT
000496  44 00 C 1D0                 EX    0,HOOK..IF-FALSE


* STATEMENT NUMBER  44
00049A                       CL.17  EQU   *
00049A  44 00 C 1AC                 EX    0,HOOK..STMT
00049E  41 70 D 108                 LA    7,264(0,13)
0004A2  50 70 3 248                 ST    7,584(0,3)
0004A6  D2 07 D 454 3 138           MVC   1108(8,13),312(3)
0004AC  41 80 D 432                 LA    8,1074(0,13)
0004B0  50 80 D 454                 ST    8,1108(0,13)
0004B4  41 70 D 454                 LA    7,1108(0,13)
0004B8  50 70 3 24C                 ST    7,588(0,3)
0004BC  96 80 3 24C                 OI    588(3),X'80'
0004C0  18 5D                       LR    5,13
0004C2  41 10 3 248                 LA    1,584(0,3)
0004C6  58 F0 3 02C                 L     15,A..NEXT_WORD
0004CA  44 00 C 1C0                 EX    0,HOOK..PRE-CALL
0004CE  47 01 4 00A                 NOP   HOOK..INFO
0004D2  05 EF                       BALR  14,15
0004D4  44 00 C 1C4                 EX    0,HOOK..POST-CALL
0004D8  D2 00 D 1EC D 432           MVC   WORD(1),1074(13)
0004DE  48 F0 D 432                 LH    15,1074(0,13)
0004E2  44 F0 2 19E                 EX    15,CL.72
0004E6  47 F0 2 1A4                 B     CL.73
0004EA                       CL.72  EQU   *
0004EA  D2 00 D 1ED D 433           MVC   WORD+1(1),1075(13)
0004F0                       CL.73  EQU   *


* STATEMENT NUMBER  45
0004F0  44 00 C 1AC                 EX    0,HOOK..STMT
0004F4  47 F0 2 0D0                 B     CL.48
0004F8                       CL.15  EQU   *
```

Object listing. This is a partial listing of the machine instructions generated by the compiler from the PL/I source program.

1 Machine instructions in hexadecimal

2 Assembler-language form of the machine instruction

3 HOOK indicates a location where the debugging routines can get control. These are included because the program was compiled with the TEST option.

```
5686-069 IBM PL/I for VSE/ESA          /* PL/I Sample Program: Used to verify product installation          */ PAGE  44
COMPILER DIAGNOSTIC MESSAGES
  1     2   3
ERROR ID L   STMT    MESSAGE DESCRIPTION

COMPILER INFORMATORY MESSAGES

IEL0533I I           NO 'DECLARE' STATEMENT(S) FOR 'INDEX'.
IEL0871I I   62      RESULT OF BUILTIN FUNCTION 'SUM' WILL BE EVALUATED USING FIXED POINT ARITHMETIC OPERATIONS.

END OF COMPILER DIAGNOSTIC MESSAGES
  4                                    5
COMPILE TIME    0.20 MINS        SPILL FILE:     0 RECORDS, SIZE  4051
END OF COMPILATION OF SAMPLE
```

Diagnostic messages and an end-of-compile-step message generated by the compiler. All diagnostic messages generated by the compiler are documented in the *PL/I VSE Compile-Time Messages and Codes* book.

1. ERROR ID identifies the message as originating from the PL/I compiler (IEL), and gives the message number.
2. L indicates the severity level of the message.
3. STMT gives the number of the statement in which the error occurred.
4. Compile time in minutes. This time includes the preprocessor.
5. Number of records "spilled" into auxiliary storage and the size in bytes of the spill file records.

```
JOB  IEL1EIVP  10/27/94   5686-066-06-15C-0   LINKAGE EDITOR DIAGNOSTIC OF INPUT
ACTION TAKEN  MAP       [1]                                                  [2]
            ** MODULE CEEBETBL 94-08-10 15.37 94-09-13 17.32  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBINT  94-08-10 15.37 94-09-13 17.32  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBLLST 94-08-10 15.37 94-09-13 17.32  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEBTRM  94-08-10 15.37 94-09-28 15.59  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEROOTA 94-08-10 15.37 94-09-28 16.05  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEARLU  94-08-10 15.37 94-09-13 17.32  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEEINT   94-08-10 15.37 94-09-13 17.32  AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
INCLUDE CEEBPIRA
            ** MODULE CEEBPIRA 94-08-29 16.39 94-09-13 17.32  INCLUDED FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE CEESG010 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE CEEPMATH 94-09-15 11.00                 AUTOLNKD FROM  PRD2    .SCEEBASE  VOLID=PRDUCT
            ** MODULE IBMRINP1 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSASCA 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEDB 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEDF 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEDX 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEFX 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEZB 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEZF 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCEZX 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHFD 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHFE 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHFH 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHFP 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHFY 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHXD 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHXE 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHXF 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHXH 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHXP 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCHXY 94-10-21 15.16                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCWDH 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSCWZH 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSEOCA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSJDSA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSJDSB 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSOCLA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSOCLB 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSOCLC 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSOCLD 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSRIOA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSRIOB 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSRIOC 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSRIOD 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSEOA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSIOA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSIOB 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSIOC 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSIOD 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSIOE 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSIOT 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSLOA 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSLOB 94-10-21 15.20                 AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
```

Linkage editor listing part 1 - input list.

1 Name of each object module included.

2 Name of the sublibrary where each object module was found.

```
JOB  IEL1EIVP  10/27/94   5686-066-06-15C-0   LINKAGE EDITOR DIAGNOSTIC OF INPUT
            ** MODULE IBMSSPLA 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSPLB 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSPLC 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSXCA 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSXCB 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSXCC 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
            ** MODULE IBMSSXCD 94-10-21 15.20                AUTOLNKD FROM  PRD2    .PROD      VOLID=PRDUCT
ENTRY
10/27/94 PSEUDOREGISTERS:                  NAME      ORIGIN        LENGTH
                                           SOURCE         0             4
         TOTAL LENGTH OF PSEUDOREGISTERS:                             4
10/27/94 PHASE    XFR-AD  LOCORE  HICORE    CSECT/    LOADED   RELOC.  PARTIT.  PHASE   TAKEN  AMODE/RMODE
                                             ENTRY     AT       FACTOR  OFFSET   OFFSET  FROM       2
         ------------------------------------------------------------------------------------------------------
         PHASE*** 800078  800078  8039EB                                                    1       31 ANY  RELOCATABLE
         ------------------------------------------------------------------------------------------------------
                                            CEESTART  800078   800078  000000   000000  SYSLNK   ANY ANY
                                             *SAMPLE1  8000F8   8000F8  000080   000080  SYSLNK   ANY ANY
                                            +SAMPLE    800100
                                             *SAMPLE2  801240   801240  0011C8   0011C8  SYSLNK   ANY ANY
                                             CEEMAIN   801D88   801D88  001D10   001D10  SYSLNK   ANY ANY
                                             IELCGOG   801D98   801D98  001D20   001D20  SYSLNK   ANY ANY
                                             IELCGOH   801E48   801E48  001DD0   001DD0  SYSLNK   ANY ANY
                                             IELCGOC   801EE8   801EE8  001E70   001E70  SYSLNK   ANY ANY
                                             IELCGMY   801F68   801F68  001EF0   001EF0  SYSLNK   ANY ANY
                                             IELCGCY   802010   802010  001F98   001F98  SYSLNK   ANY ANY
                                             CEEUOPT   802090   802090  002018   002018  SYSLNK   ANY ANY
                                             PLIXOPT   802568   802568  0024F0   0024F0  SYSLNK   ANY ANY
                                             PLIXOPT*  8025D0   8025D0  002558   002558  SYSLNK   ANY ANY
                                            *PLIXOPT+  8025D4
                                            +PLIXOPT-  8025F0
                                             SAMPLE*   802600   802600  002588   002588  SYSLNK   ANY ANY
                                            *SAMPLE+   802600
                                             SOURCE    802620   802620  0025A8   0025A8  SYSLNK   ANY ANY
                                             SOURCE*   802688   802688  002610   002610  SYSLNK   ANY ANY
                                            *SOURCE+   802688
                                             SYSPINT   8026A8   8026A8  002630   002630  SYSLNK   ANY ANY
                                             SYSPINT*  802718   802718  0026A0   0026A0  SYSLNK   ANY ANY
                                            *SYSPINT+  802718
                                             CEEBETBL  802738   802738  0026C0   0026C0  CEEBETBL ANY ANY
                                             CEEBINT   802758   802758  0026E0   0026E0  CEEBINT  ANY ANY
                                             CEEBLLST  802760   802760  0026E8   0026E8  CEEBLLST ANY ANY
                                            *CEELLIST  802770
                                             CEEBTRM   8027C0   8027C0  002748   002748  CEEBTRM  ANY ANY
                                             CEEROOTA  802B68   802B68  002AF0   002AF0  CEEROOTA ANY ANY
                                             CEEARLU   8031F0   8031F0  003178   003178  CEEARLU  ANY ANY
                                             CEEBPIRA  803328   803328  0032B0   0032B0  CEEBPIRA ANY ANY
                                            +CEEINT    803328
                                            *CEEBPIRB  803328
                                            *CEEBPIRC  803328
                                             CEESG010  8034C8   8034C8  003450   003450  CEESG010 ANY ANY
                                             CEEPMATH  803530   803530  0034B8   0034B8  CEEPMATH ANY ANY
                                            *IBMSMATH  803530
                                             IBMRINP1  803548   803548  0034D0   0034D0  IBMRINP1 ANY ANY
                                             IBMSASCA  803570   803570  0034F8   0034F8  IBMSASCA ANY ANY
                                            *IBMBASCA  803570
                                             IBMSCEDB  803588   803588  003510   003510  IBMSCEDB ANY ANY
                                            *IBMBCEDB  803588
                                             IBMSCEDF  8035A0   8035A0  003528   003528  IBMSCEDF ANY ANY
                                            *IBMBCEDF  8035A0
                                             IBMSCEDX  8035B8   8035B8  003540   003540  IBMSCEDX ANY ANY
                                            *IBMBCEDX  8035B8
                                             IBMSCEFX  8035D0   8035D0  003558   003558  IBMSCEFX ANY ANY
                                            *IBMBCEFX  8035D0
                                             IBMSCEZB  8035E8   8035E8  003570   003570  IBMSCEZB ANY ANY
```

Linkage editor listing part 2 - map of the resultant phase.

This is a list of all CSECTS in the phase, with the offset of each CSECT shown.

1 Each CSECT is either taken from SYSLNK (compiled in this job) or a library object member.

2 Each module has its own addressing mode (AMODE) and residency mode (RMODE). The AMODE and RMODE of the phase will be determined by the mode of the modules.

```
10/27/94 PHASE    XFR-AD  LOCORE  HICORE    CSECT/    LOADED   RELOC.  PARTIT. PHASE   TAKEN  AMODE/RMODE
                                            ENTRY     AT       FACTOR  OFFSET  OFFSET  FROM
                                          *IBMBCEZB  8035E8
                                           IBMSCEZF  803600   803600  003588  003588  IBMSCEZF ANY ANY
                                          *IBMBCEZF  803600
                                           IBMSCEZX  803618   803618  0035A0  0035A0  IBMSCEZX ANY ANY
                                          *IBMBCEZX  803618
                                           IBMSCHFD  803630   803630  0035B8  0035B8  IBMSCHFD ANY ANY
                                          *IBMBCHFD  803630
                                           IBMSCHFE  803648   803648  0035D0  0035D0  IBMSCHFE ANY ANY
                                          *IBMBCHFE  803648
                                           IBMSCHFH  803660   803660  0035E8  0035E8  IBMSCHFH ANY ANY
                                          *IBMBCHFH  803660
                                           IBMSCHFP  803678   803678  003600  003600  IBMSCHFP ANY ANY
                                          *IBMBCHFP  803678
                                           IBMSCHFY  803690   803690  003618  003618  IBMSCHFY ANY ANY
                                          *IBMBCHFY  803690
                                           IBMSCHXD  8036A8   8036A8  003630  003630  IBMSCHXD ANY ANY
                                          *IBMBCHXD  8036A8
                                           IBMSCHXE  8036C0   8036C0  003648  003648  IBMSCHXE ANY ANY
                                          *IBMBCHXE  8036C0
                                           IBMSCHXF  8036D8   8036D8  003660  003660  IBMSCHXF ANY ANY
                                          *IBMBCHXF  8036D8
                                           IBMSCHXH  8036F0   8036F0  003678  003678  IBMSCHXH ANY ANY
                                          *IBMBCHXH  8036F0
                                           IBMSCHXP  803708   803708  003690  003690  IBMSCHXP ANY ANY
                                          *IBMBCHXP  803708
                                           IBMSCHXY  803720   803720  0036A8  0036A8  IBMSCHXY ANY ANY
                                          *IBMBCHXY  803720
                                           IBMSCWDH  803738   803738  0036C0  0036C0  IBMSCWDH ANY ANY
                                          *IBMBCWDH  803738
                                           IBMSCWZH  803750   803750  0036D8  0036D8  IBMSCWZH ANY ANY
                                          *IBMBCWZH  803750
                                           IBMSEOCA  803768   803768  0036F0  0036F0  IBMSEOCA ANY ANY
                                          *IBMBEOCA  803768
                                           IBMSJDSA  803780   803780  003708  003708  IBMSJDSA ANY ANY
                                          *IBMBJDSA  803780
                                           IBMSJDSB  803798   803798  003720  003720  IBMSJDSB ANY ANY
                                          *IBMBJDSB  803798
                                           IBMSOCLA  8037B0   8037B0  003738  003738  IBMSOCLA ANY ANY
                                          *IBMBOCLA  8037B0
                                           IBMSOCLB  8037C8   8037C8  003750  003750  IBMSOCLB ANY ANY
                                          *IBMBOCLB  8037C8
                                           IBMSOCLC  8037E0   8037E0  003768  003768  IBMSOCLC ANY ANY
                                          *IBMBOCLC  8037E0
                                           IBMSOCLD  8037F8   8037F8  003780  003780  IBMSOCLD ANY ANY
                                          *IBMBOCLD  8037F8
                                           IBMSRIOA  803810   803810  003798  003798  IBMSRIOA ANY ANY
                                          *IBMBRIOA  803810
                                           IBMSRIOB  803828   803828  0037B0  0037B0  IBMSRIOB ANY ANY
                                          *IBMBRIOB  803828
                                           IBMSRIOC  803840   803840  0037C8  0037C8  IBMSRIOC ANY ANY
                                          *IBMBRIOC  803840
                                           IBMSRIOD  803858   803858  0037E0  0037E0  IBMSRIOD ANY ANY
                                          *IBMBRIOD  803858
```

```
10/27/94 PHASE    XFR-AD  LOCORE  HICORE    CSECT/    LOADED   RELOC.  PARTIT. PHASE   TAKEN  AMODE/RMODE
                                            ENTRY     AT       FACTOR  OFFSET  OFFSET  FROM
                                             IBMSSEOA  803870   803870  0037F8  0037F8  IBMSSEOA ANY ANY
                                            *IBMBSEOA  803870
                                             IBMSSIOA  803888   803888  003810  003810  IBMSSIOA ANY ANY
                                            *IBMBSIOA  803888
                                             IBMSSIOB  8038A0   8038A0  003828  003828  IBMSSIOB ANY ANY
                                            *IBMBSIOB  8038A0
                                             IBMSSIOC  8038B8   8038B8  003840  003840  IBMSSIOC ANY ANY
                                            *IBMBSIOC  8038B8
                                             IBMSSIOD  8038D0   8038D0  003858  003858  IBMSSIOD ANY ANY
                                            *IBMBSIOD  8038D0
                                             IBMSSIOE  8038E8   8038E8  003870  003870  IBMSSIOE ANY ANY
                                            *IBMBSIOE  8038E8
                                             IBMSSIOT  803900   803900  003888  003888  IBMSSIOT ANY ANY
                                            *IBMBSIOT  803900
                                             IBMSSLOA  803918   803918  0038A0  0038A0  IBMSSLOA ANY ANY
                                            *IBMBSLOA  803918
                                             IBMSSLOB  803930   803930  0038B8  0038B8  IBMSSLOB ANY ANY
                                            *IBMBSLOB  803930
                                             IBMSSPLA  803948   803948  0038D0  0038D0  IBMSSPLA ANY ANY
                                            *IBMBSPLA  803948
                                             IBMSSPLB  803960   803960  0038E8  0038E8  IBMSSPLB ANY ANY
                                            *IBMBSPLB  803960
                                             IBMSSPLC  803978   803978  003900  003900  IBMSSPLC ANY ANY
                                            *IBMBSPLC  803978
                                             IBMSSXCA  803990   803990  003918  003918  IBMSSXCA ANY ANY
                                            *IBMBSXCA  803990
                                             IBMSSXCB  8039A8   8039A8  003930  003930  IBMSSXCB ANY ANY
                                            *IBMBSXCB  8039A8
                                             IBMSSXCC  8039C0   8039C0  003948  003948  IBMSSXCC ANY ANY
                                            *IBMBSXCC  8039C0
                                             IBMSSXCD  8039D8   8039D8  003960  003960  IBMSSXCD ANY ANY
                                            *IBMBSXCD  8039D8
UNRESOLVED EXTERNAL REFERENCES                       WXTRN     CEEFMAIN
                                                     WXTRN     IBMSSIST
                                                     WXTRN     CEEBXITA
                                                     WXTRN     CEESG000
                                                     WXTRN     CEESG001
                                                     WXTRN     CEESG002
                                                     WXTRN     CEESG003
                                                     WXTRN     CEESG004
                                                     WXTRN     CEESG005
                                                     WXTRN     CEESG006
                                                     WXTRN     CEESG007
                                                     WXTRN     CEESG008
                                                     WXTRN     CEESG009
                                                     WXTRN     CEESG011
                                                     WXTRN     CEESG012
                                                     WXTRN     CEESG013
                                                     WXTRN     CEESG014
                                                     WXTRN     CEESG015
                                                     WXTRN     CEESG016
                                                     WXTRN     PLISTART
```

Unresolved external references.

External references that the linkage editor was not able to resolve.

WXTRN denotes a weak external reference. This will give a return code of 2.

EXTRN denotes a full external reference, and will give a return code of 4.

```
10/27/94 PHASE    XFR-AD  LOCORE  HICORE    CSECT/     LOADED    RELOC.  PARTIT. PHASE   TAKEN  AMODE/RMODE
                                             ENTRY       AT       FACTOR  OFFSET  OFFSET  FROM
                                                    WXTRN     IBMBPOPT
                                                    WXTRN     PLITABS
                                                    WXTRN     PLIMAIN

UNRESOLVED ADCON  AT OFFSET 008000E0
UNRESOLVED ADCON  AT OFFSET 00801EE0
UNRESOLVED ADCON  AT OFFSET 00801F60
UNRESOLVED ADCON  AT OFFSET 0080273C
UNRESOLVED ADCON  AT OFFSET 00802770
UNRESOLVED ADCON  AT OFFSET 00802774
UNRESOLVED ADCON  AT OFFSET 00802778
UNRESOLVED ADCON  AT OFFSET 0080277C
UNRESOLVED ADCON  AT OFFSET 00802780
UNRESOLVED ADCON  AT OFFSET 00802784
UNRESOLVED ADCON  AT OFFSET 00802788
UNRESOLVED ADCON  AT OFFSET 0080278C
UNRESOLVED ADCON  AT OFFSET 00802790
UNRESOLVED ADCON  AT OFFSET 00802794
UNRESOLVED ADCON  AT OFFSET 0080279C
UNRESOLVED ADCON  AT OFFSET 008027A0
UNRESOLVED ADCON  AT OFFSET 008027A4
UNRESOLVED ADCON  AT OFFSET 008027A8
UNRESOLVED ADCON  AT OFFSET 008027AC
UNRESOLVED ADCON  AT OFFSET 008027B0
UNRESOLVED ADCON  AT OFFSET 00803118
UNRESOLVED ADCON  AT OFFSET 00803518
UNRESOLVED ADCON  AT OFFSET 00803514
UNRESOLVED ADCON  AT OFFSET 00803500
UNRESOLVED ADCON  AT OFFSET 008034E8
UNRESOLVED ADCON  AT OFFSET 008034D8
026 UNRESOLVED ADDRESS CONSTANTS

 ************************
 *** Word-use Report ***  1
 ************************
 -count-   --word--
     3       BEGIN
     1       CLOSE
    13       DCL
    24       DECLARE
     2       DISPLAY
    14       DO
    13       ELSE
    23       END
     1       GO
    13       IF
------------The previous value should have been    14    2
     7       LIST
     4       ON
     1       OPEN
     2       PROC
     3       PROCEDURE
     2       READ
     4       RETURN
     1       SELECT
     2       STOP
    13       THEN
     2       WHEN
There were            148 references to    36 words.
There was a discrepancy in at least one of the word-counts.  2
```

Sample program output

1 Program output header

2 The apparent error is intentional

Bibliography

IBM PL/I for VSE/ESA publications

Fact Sheet, GC26-8052

Installation and Customization Guide, SC26-8057

Licensed Program Specifications, GC26-8055

Language Reference, SC26-8054

Compile-Time Messages and Codes, SC26-8059

Diagnosis Guide, SC26-8058

Migration Guide, SC26-8056

Programming Guide, SC26-8053

Reference Summary, SX26-3836

IBM Language Environment for VSE/ESA publications

Concepts Guide, GC26-8063

Fact Sheet, GC26-8062

Debugging Guide and Run-Time Messages, SC26-8066

Diagnosis Guide, SC26-8060

Installation and Customization Guide, SC26-8064

Licensed Program Specifications, GC26-8061

Programming Guide, SC26-8065

Reference Summary, SX26-3835

VSE/ESA publications

VSE/ESA Version 1

Administration, SC33-6505

Messages and Codes, SC33-6507

System Control Statements, SC33-6513

System Utilities, SC33-6517

System Macros Reference, SC33-6516

Guide to System Functions, SC33-6511

VSE/VSAM Commands and Macros, SC33-6532

VSE/VSAM User's Guide, SC33-6535

VSE/ESA Version 2

Administration, SC33-6605

Messages and Codes, SC33-6607

System Control Statements, SC33-6613

System Utilities, SC33-6617

System Macros Reference, SC33-6616

Guide to System Functions, SC33-6611

VSE/VSAM Commands and Macros, SC33-6631

VSE/VSAM User's Guide, SC33-6632

Related publications

CICS/VSE

Application Programming Guide, SC33-0712

Application Programming Reference, SC33-0713

System Definition and Operations Guide, SC33-0706

Resource Definition, SC33-0708

DFSORT for VSE/ESA

Application Programming Guide, SC26-7040

Sort/Merge II

DOS/VS VM/SP Sort/Merge Version 2 Application Programming Guide, SC33-4044

DL/I DOS/VS

Application Programming: High Level Programming Interface, SH24-5009

Application Programming: CALL and RQDLI Interfaces, SH12-5411

SQL/DS

SQL/Data System Application Programming Guide for VSE, SH09-8098

Softcopy publications

These collections contain the LE/VSE and LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

Application Development Collection, SK2T-1237

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 the *IBM Dictionary of Computing*, SC20-1699.

References: This glossary uses the following references:

Contrast with: Indicates a term or terms that have an opposed or substantially different meaning.

See: Refers to a multiple-word term in which this term appears.

See also: Refers to related terms that have similar (but not synonymous) meanings.

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.

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.

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 may 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 single quotes 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 or BX.

bit string. A string composed of zero or more bits.

bit string operators. The logical operators not (¬), 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 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.

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 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 procedure or a package block. The entire procedure, and the BEGIN statement and its corresponding END statements are not contained in the block.

containing block. The 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.

conversion. The transformation of a value from one representation to another to conform to a given set of attributes. For example, converting a character string to an arithmetic value such as FIXED BINARY (15,0).

cross section of an array. The elements represented by the extent of at least one dimension of an array. An asterisk in the place of a subscript in an array reference indicates the entire extent of that dimension.

current generation. The generation of an automatic or controlled variable that is currently available by referring to the name of the variable.

D

data. Representation of information or of value in a form suitable for processing.

data aggregate. A data item that is a collection of other data items.

data attribute. A keyword that specifies the type of data that the data item represents, such as FIXED BINARY.

data-directed transmission. The type of stream-oriented transmission in which data is transmitted. It resembles an assignment statement and is of the form:

```
name = constant
```

data item. A single named unit of data.

data list. In stream-oriented transmission, a parenthesized list of the data items used in GET and PUT statements. Contrast with *format list*.

data set. (1) A collection of data external to the program that can be accessed by reference to a single file name. (2) A device that can be referenced.

data specification. The portion of a stream-oriented transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list(s) and, for edit-directed mode, the format list(s).

data stream. Data being transferred from or to a data set by stream-oriented transmission, as a continuous stream of data elements in character form.

data transmission. The transfer of data from a data set to the program or vice versa.

data type. A set of data attributes.

DBCS. In the character set, each character is represented by two consecutive bytes.

deactivated. The state in which an identifier is said to be when its value cannot replace a preprocessor identifier in source program text. Contrast with *active*.

debugging. Process of removing bugs from a program.

decimal. The number system whose numerals are 0 through 9.

decimal digit. One of the digits 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 of 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.

enclave. In Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

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 may be invoked. See *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 may 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 upper and lower case alphabetic characters A through Z, $, @, and #.

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. A procedure that is not contained in any other procedure.

extralingual character. Characters (such as $, @, and #) that are not classified as alphanumeric or special.

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*.

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 half-word, full-word, or double-word (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 non-zero 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.

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.

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 (&), and or (|).

loop. A sequence of instructions that is executed iteratively.

lower bound. The lower limit of an array dimension.

M

main procedure. An external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. This procedure is invoked automatically as the first step in the execution of a program.

major structure. A structure whose name is declared with level number 1.

member. A structure, union, or element name, possibly dimensioned, in a structure or union.

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 may 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. 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 may be used to influence the execution or interpretation of the statement.

P

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.

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.

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 may be followed by an argument list. It may appear in a CALL statement or the CALL option, or as a function reference.

process. The highest level of the Language Environment program management model. A process is a collection of resources, both program code and data, and consists of at least one enclave.

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 may 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). The portion of a program throughout which a particular name is known.

scope (of a name). See *scope (of a declaration)*.

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*.

sign and currency symbol characters. The picture specification characters S, +, –, and $.

simple parameter. A parameter for which no storage class attribute is specified. It may 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 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*.

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 may 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.

synchronous. A single flow of control for serial execution of a program.

T

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.

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.

thread. The basic run-time path within the Language Environment program management model. It is dispatched by the system with its own instruction counter and registers. The thread is where actual code resides.

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 considered to be in error.

upper bound. The upper limit of an array dimension.

V

value reference. A reference used to obtain the value of an item of data.

variable. A named entity used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times.

variable reference. A reference that designates all or part of a variable.

virtual origin (VO). The location where the element of the array whose subscripts are all zero are held. If such an element does not appear in the array, the virtual origin is where it would be held.

Z

zero-suppression characters. The picture specification characters Z and *, which are used to suppress zeros in the corresponding digit positions and replace them with blanks or asterisks respectively.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

We'd Like to Hear from You

IBM PL/I for VSE/ESA
Programming Guide
Release 1

Publication No. SC26-8053-00

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 PL/I for VSE/ESA
Programming Guide
Release 1

Publication No. SC26-8053-00

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	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

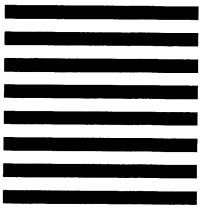
Company or Organization

Phone No.

Readers' Comments
SC26-8053-00

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

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape | **Please do not staple** | Fold and Tape

Readers' Comments

IBM PL/I for VSE/ESA
Programming Guide
Release 1

Publication No. SC26-8053-00

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	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name _____ Address _____

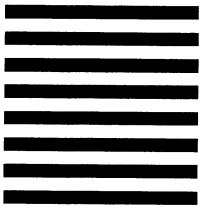
Company or Organization _____

Phone No. _____

Readers' Comments
SC26-8053-00

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

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape | **Please do not staple** | Fold and Tape

Program Number: 5686-069

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

IBM PL/I for VSE/ESA Publications

GC26-8052	Fact Sheet
GC26-8055	Licensed Program Specifications
SC26-8056	Migration Guide
SC26-8057	Installation and Customization Guide
SC26-8053	Programming Guide
SC26-8054	Language Reference
SX26-3836	Reference Summary
SC26-8058	Diagnosis Guide
SC26-8059	Compile-Time Messages and Codes

SC26-8053-00